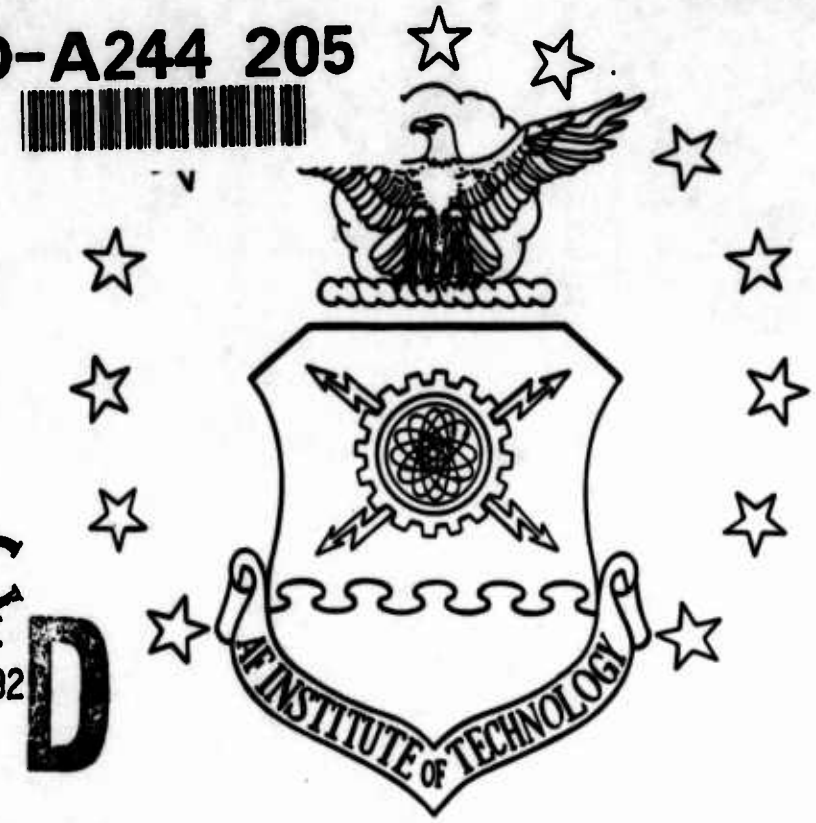
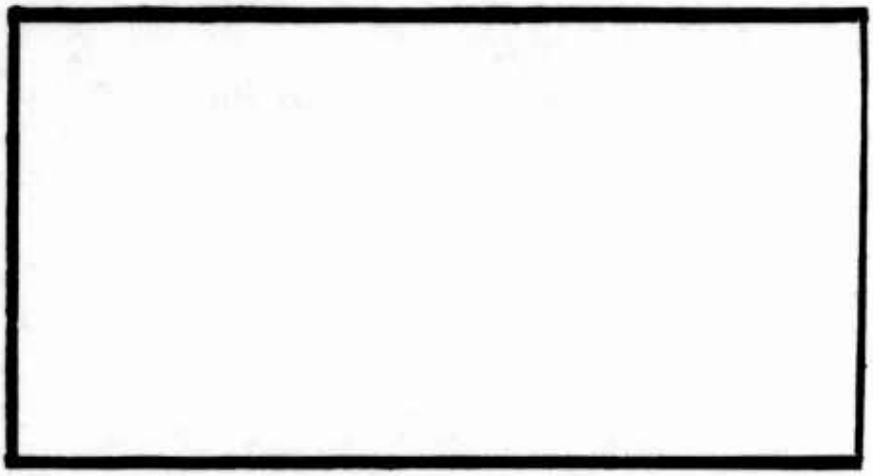




DEPARTMENT OF THE AIR FORCE

AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

AFIT/GCS/ENG/91D-12

GRAPH-BASED VISUALIZATION OF FORMAL SPECIFICATION AND DOMAIN SPECIFIC LANGUAGES

THESIS

Randel Kevin Langloss
Captain, USAF

AFIT/GCS/ENG/91D-12

AFIT/GCS/ENG/91D-12

GRAPH-BASED VISUALIZATION OF FORMAL SPECIFICATION AND DOMAIN SPECIFIC LANGUAGES

THESIS

Presented to the Faculty of the School of Engineering
of the Air Force Institute of Technology
Air University
In Partial Fulfillment of the
Requirements for the Degree of
Master of Science (Computer Science)

Randel Kevin Langloss, A.S., B.S.
Captain, USAF

December, 1991

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	December 1991	Master's Thesis

4. TITLE AND SUBTITLE

GRAPH-BASED VISUALIZATION OF FORMAL SPECIFICATION AND DOMAIN SPECIFIC LANGUAGES

5. FUNDING NUMBERS

6. AUTHOR(S)

Randel K. Langloss, Capt, USAF

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Air Force Institute of Technology, WPAFB OH 45433-6583

8. PERFORMING ORGANIZATION REPORT NUMBER

AFIT/GCS/ENG/91D-12

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

ROME LABORATORIES
RL/C3CA
GRIFFISS AFB, NY 13441-5700
PH: (315)330-3564 DSN: 587-3564

10. SPONSORING / MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION / AVAILABILITY STATEMENT

Approved for public release; distribution unlimited

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

This research develops and implements Visual Refine, a graph-based visualization system, for the Refine™ wide-spectrum formal specification language and environment developed and marketed by Reasoning Systems, Inc. Refine specifications are represented in the Refine object base as abstract syntax trees (AST). Using these AST representations, one-to-one mappings are defined between nodes of the AST and the graphical icons of Visual Refine. Visual Refine uses these mappings to implement a set of formal transformations. Each transformation is encapsulated within a Refine rule, and this set of rules form the Visual Refine transformation system. The Visual Refine transformation system, in conjunction with Refine object base manipulation facilities, is then used to create graph-based visualizations of Refine specifications. To illustrate that visualization aids in the understanding of a formal specification, Visual Refine is applied to an example Refine specification. Finally, it is shown that the technology developed for Visual Refine is general enough to be applied to any language that can be represented as an AST in the Refine object base. Specifically, the technology is shown to be applicable and beneficial towards formalizing Domain Specific Software Architectures.

14. SUBJECT TERMS

Graphics, Interactive Graphics, Knowledge Based Systems, Language, Programming Languages, Language Translation, Machine Translation, Specifications, Software Engineering

15. NUMBER OF PAGES

230

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

Preface

The research effort required to accomplish this thesis has served two main purposes. First, it has enabled me to realize the tremendous capabilities, and awesome potential, that formal methods provide to the field of software engineering. Second, and perhaps more importantly, this research has provided the Air Force Institute of Technology (AFIT), formal methods research group, with the necessary understanding of the Refine^{TM} formal specification environment, including its object base, specification language, interface capabilities, and powerful language processing capabilities. By understanding these capabilities, future researchers will be able to develop sophisticated Domain Models and formal object specification languages. Using the visualization technology developed in this thesis, future researchers will be able to better understand the complicated interactions between the objects they specify. My hope and prayer is that this research effort is found to be useful by those who follow after.

I would like to give thanks and credit, first and most importantly, to Jesus Christ. Without Him nothing that I might have accomplished would make any difference. I would also like to thank my advisor, Major Paul Bailor, for his patience and guidance in conducting this research. Additionally, I would like to thank the members of my research committee, Lieutenant Colonel Elton P. Amburn and Major Gregg Gunch, for their diligence in reading and commenting on this thesis. Finally, I would like to thank my wife [redacted] daughter [redacted] and sons [redacted], for their never ending support, encouragement, sacrifices, and prayers during the course of this research effort.

Randel Kevin Langloss

Table of Contents

List of Figures

List of Tables

AFIT/GCS/ENG/91D-12

Abstract

This research develops and implements Visual Refine, a graph-based visualization system, for the Refine^{TM} wide-spectrum formal specification language and environment developed and marketed by Reasoning Systems, Inc. Refine specifications are represented in the Refine object base as abstract syntax trees (AST). Using these AST representations, one-to-one mappings are defined between nodes of the AST and the graphical icons of Visual Refine. Visual Refine uses these mappings to implement a set of formal transformations. Each transformation is encapsulated within a Refine rule, and this set of rules form the Visual Refine transformation system. The Visual Refine transformation system, in conjunction with Refine object base manipulation facilities, is then used to create graph-based visualizations of Refine specifications. To illustrate that visualization aids in the understanding of a formal specification, Visual Refine is applied to an example Refine specification. Finally, it is shown that the technology developed for Visual Refine is general enough to be applied to any language that can be represented as an AST in the Refine object base. Specifically, the technology is shown to be applicable and beneficial towards formalizing Domain Specific Software Architectures.

Graph-Based Visualization of Formal Specification and Domain Specific Languages

I. Introduction

1.1 Background

Computer software has infiltrated every aspect of modern life. And yet, virtually every computer program ever written contains errors, anomalies, or peculiarities (11:3). These errors, commonly known as "bugs," can not simply be attributed to the people building modern software systems; rather, these bugs are more likely due to the inadequacy of the methods used to specify, design, and implement complex software systems (11:3). The need for improvements in software development is well established (8:2 - 8), (9:547), (10:1 - 7), and (20:2 - 4); and yet, improvements in the areas of project management, requirements analysis, top-down problem decomposition, structured programming, abstract data types, information hiding, program walk-throughs, and rigorous testing have not provided the technology necessary to ensure that a program will function in the manner specified by the software designer; simply put, program correctness remains a problem (11:2 - 3). The quality of software has gotten better, as a result of the above mentioned improvements, but as the following list clearly shows, traditional methods of developing software does not, and can not, ensure that software will perform as expected when delivered (11:4):

- Space shuttle Discovery flew upside down during a laser-beam missile defense experiment due to a discrepancy in the units of expected data.
- The first version of the F-16 navigation software improperly handled crossing the equator, causing the aircraft to fly upside down.

- One version of the Apollo 11 software had the moon's gravity appearing repulsive rather than attractive.
- A computer bug caused a US warship's gun to fire opposite the intended direction during an exercise off San Francisco in 1982.
- A series of accidental radiation overdoses was administered by cancer therapy machines in Georgia because of an error in the controlling computer program.

Some might argue that more complete testing of software systems is the answer. However, exhaustive testing of even small programs is not practical. Simple mathematics show that even small programs (fifty conditional branches) would require over thirty-five years to exhaustively test on a machine capable of executing one million instructions per second (ips).

$$\frac{2^{50}\ paths}{1\ million\ ips} = 35.7 years$$

Obviously, testing is not the answer.

Mills claims that ninety percent of errors made in software development today are probably due to the immature methods used by developers, rather than to the fallibility of the people involved (14:92). With this in mind, Mills provides an excellent analogy between software of today and mathematics using Roman numerals (14:91). Until the discovery of in place notation, numbers had to be treated as whole units, rather then as a collection of individual digits. Long division could only be accomplished by highly trained mathematicians. It is likely that roman scribes had "trade secrets" for determining the quotient and remainder, similar to the trade secrets of today's programmers. By making a fundamental change in the methods used (i.e. in place notation), the problem of solving complicated long division problems was reduced to several small steps, each of which could be immediately verified. Using this methodology, it is now possible for skilled school children to solve problems beyond the capability of Euclid and Archimedes (14:91). A similar fundamental

change in the methodology used to specify, design and implement software systems must be made if the software industry is to produce reliable and correct software systems.

Boehm recognizes errors in today's software as a problem; however, he believes that economics as well as increasing requirements for software are the major motivations for improving software productivity. Boehm predicts that at a growth rate of approximately twelve percent, the software costs for the Department of Defense (DoD) will grow from the 1985 figure of $11 billion to a 1995 figure of $36 billion. with national and worldwide figures increasing to $225 billion and $450 billion respectively. Figure 1.1 is extracted from Boehm's article (7:43) to better illustrate

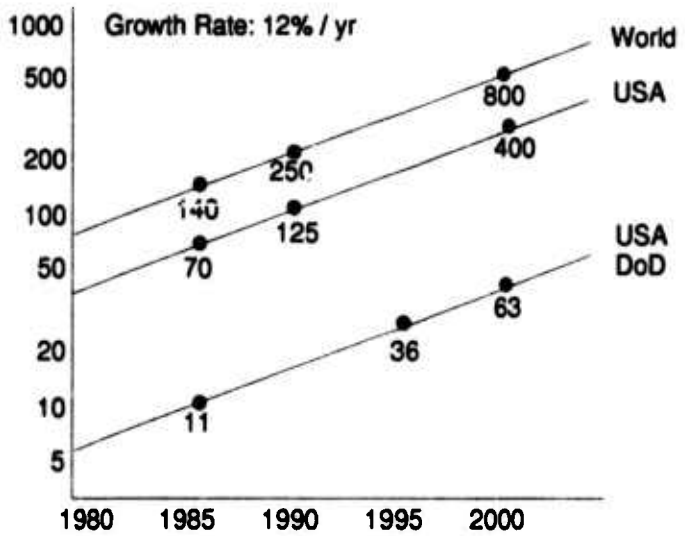


Figure 1.1. Software Cost Trends (7:figure 1)

his cost trend predictions. Besides the economic motivation, Boehm points to the backlog of software requirements as another reason to make a fundamental change in the technology used to develop modern software systems. Using the Air Force as an example, Boehm says, "the Standard Information Systems Center has identified a four-year backlog of unstarted projects representing user-validated software needs" (7:44). Simply put, industry, using current methodologies, can not produce soft-

ware at a rate equal to the demand for new products. Boehm makes the following observations (7:49):

- cost savings in software development involve
  1. making individual steps more efficient via such capabilities as automated aids to software design analysis or testing;
  2. eliminating steps via such capabilities as automatic programming or automatic quality assurance;
  3. eliminating rework via early error detection or via such capabilities as rapid prototyping to avoid later requirements rework.
- further major cost savings can be achieved by reducing the total number of elementary operation steps by developing products requiring the creation of fewer lines of code.

A fundamental change in methodology must be made in order to address the problems mentioned above. Balzer et al., in their article "Software Technology in the 1990's: Using a New Paradigm", discuss software initiatives of the Department of Defense (DoD) and the direction they should take. They compare the current software paradigm, commonly called the waterfall nodel to a new automation-based paradigm, called the transformation life cycle model. Figure 1.2 illustrates the current paradigm, upon which incremental, or evolutionary, improvements are being made (5:40). Figure 1.3 illustrates the automation based paradigm, which is based on the formal specification of software (5:40). Improvements to the current paradigm have a high degree of probability for short term payoffs; however, they are limited to the inherent weaknesses of the paradigm itself, namely, "no technology for managing the knowledge-intensive activities that constitute software development processes," and "maintenance is performed on source code" (5:39). Since the current paradigm can not affect these weaknesses, an alternative automation-based

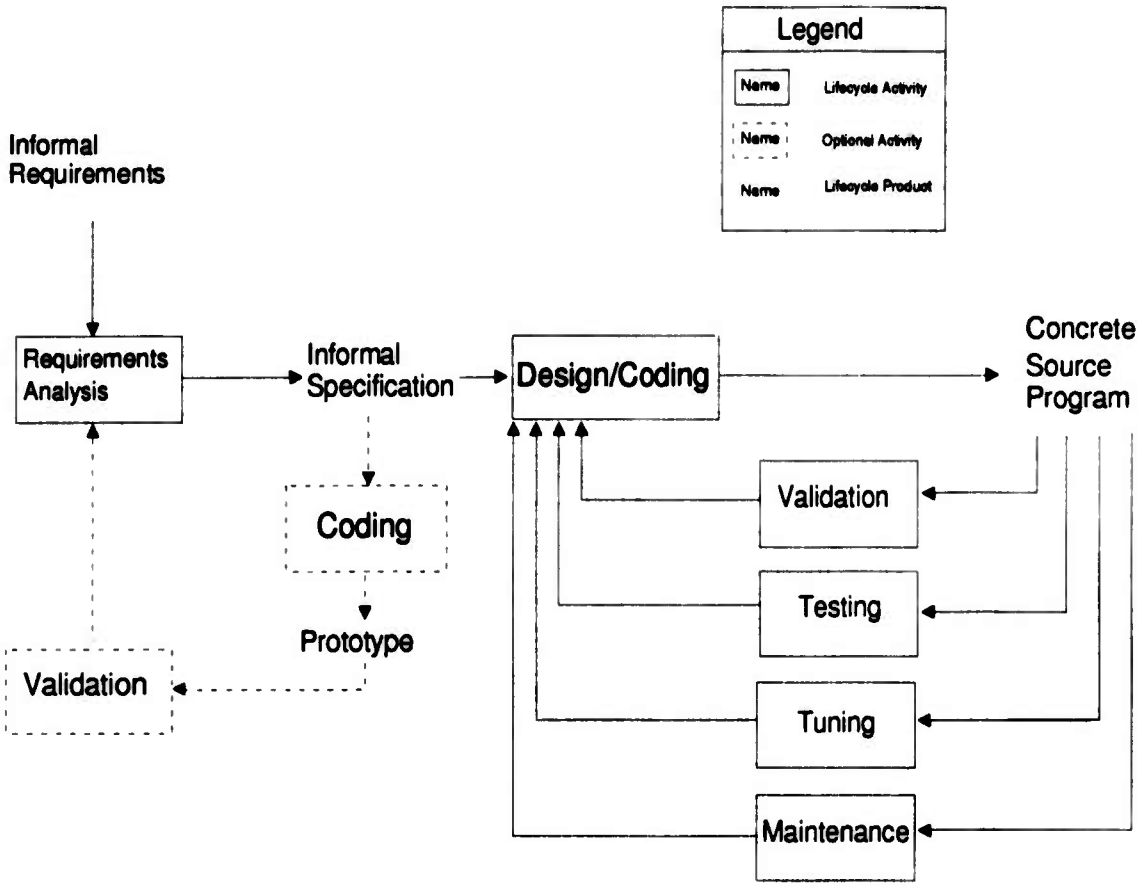


Figure 1.2. Current Paradigm: The Waterfall Model (5:figure 1(b))

paradigm is put forth in an effort to achieve order-of-magnitude improvements in software production (5:39). The new paradigm centers around formal specifications, a mathematically precise method of capturing the knowledge, and rationale, of the development process. Prototyping and maintenance activities are now accomplished early in the development life cycle (5:40) so that errors are corrected at a lower cost (7:47 - 49). Balzer et al. discuss many advantages to this new paradigm (5:41 - 44) but of importance here is the potential for vast improvements in productivity while also substantially improving software quality and correctness.

Balzer et al. clearly demonstrate the applicability of their new paradigm but fail to address the complexity commonly found in formal specifications. Formal specifications state a system's requirements in precise and unambiguous mathematical

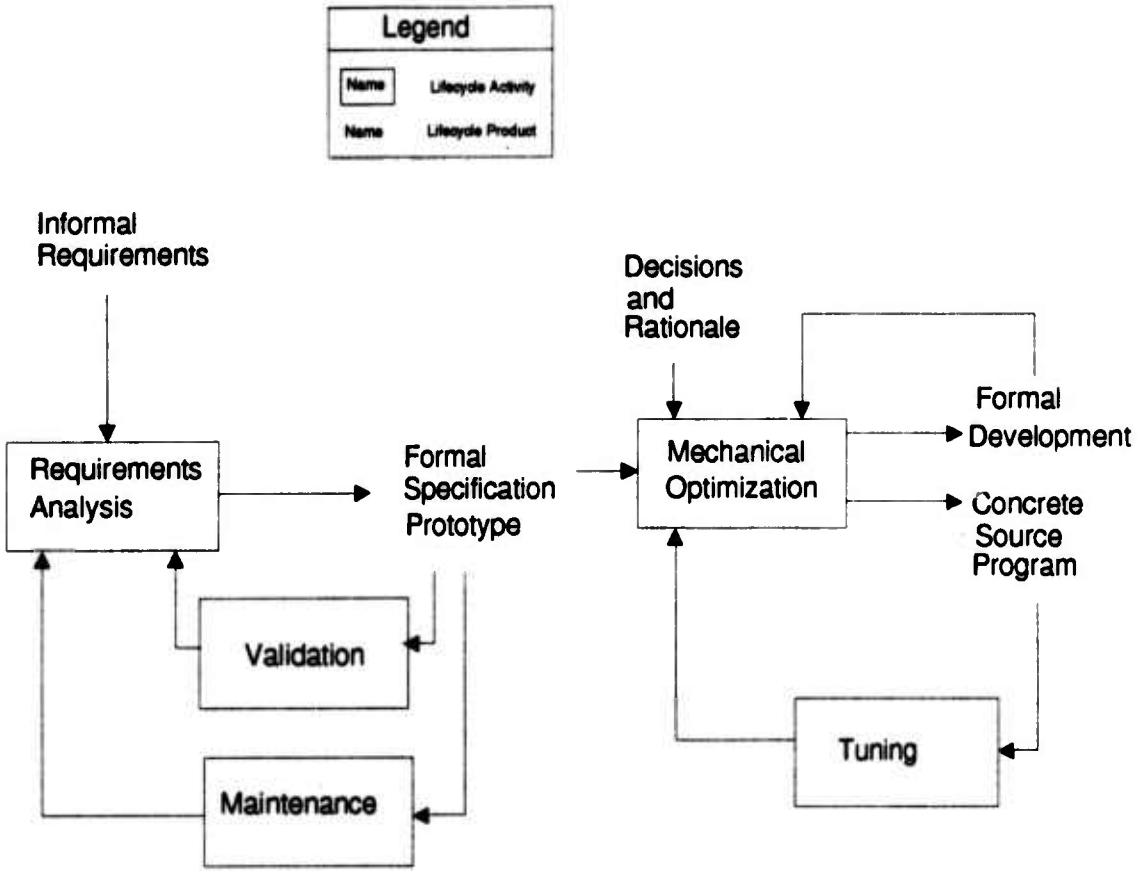


Figure 1.3. New Paradigm: Automation-Based Life Cycle Model (5:figure 1(a))

statements. Since all but the most simple software systems require complex interactions, a formal specification also becomes very complex. Additionally, most software professionals, and their users, are not adequately educated to correctly interpret the complex mathematical expressions found in a formal specification. Since the formal specification is the cornerstone upon which the benefits of Balzer's paradigm rely, the mathematical complexity, which restrain formal specifications from being widely used, must be made easier to deal with. The remaining portions of this thesis explains the construction and benefits of tools and methodologies which help to reduce the complexity of formal specifications. More specifically, this thesis demonstrates how visualizing a wide-spectrum, formal specification language improves user understanding and how this same visualization technology can be used to visualize

domain specific languages, providing greater analyst and client understanding of the problem.

1.2 Problem

1.2.1 Big Picture The Air Force Institute of Technology (AFIT), School of Engineering, Department of Electrical and Computer Engineering, Formal Methods Group has an overall research objective to "investigate/develop new methods and prototype tools that contribute to the construction of a formal foundation for software engineering", with a specific objective of "formalization of requirements analysis [domain modeling], specification, and design" (3). A formal environment for the development of software is currently being prototyped in support of the above objectives. To be of greatest use, this environment must be able to import information from the informal development methods in widespread use today. AFIT's formal environment requires an import facility so that current software projects have a mechanism to transform informal specifications and designs into formal specifications and designs. This environment must also support the formal development of software systems from the ground up. Since informal models typically represent information in a graphical manner, and since humans communicate more easily through visual or graphical means (2:2-7), AFIT's formal environment will use a formal graphical language to represent both syntax and semantics of the desired software system. Once a software developer is satisfied with a system's formal graphical representation, the environment will automatically, or at least semi-automatically, produce an executable formal specification derived by compiling the graphical specification. Figure 1.4 illustrates the relationship between AFIT's formal environment and other external components.

1.2.2 Major Subdivisions

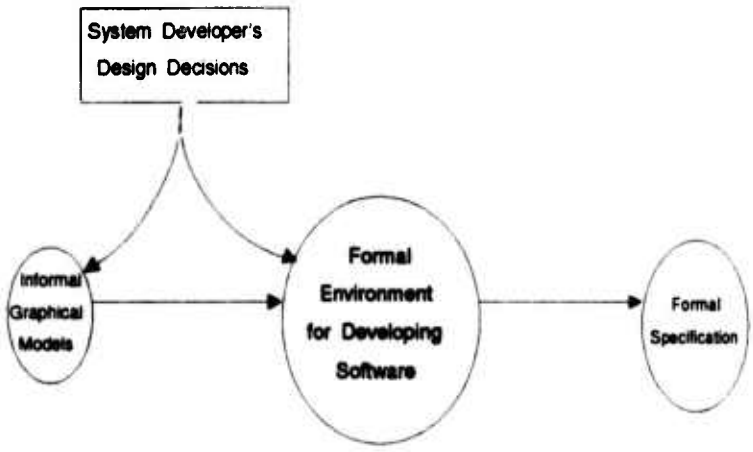


Figure 1.4. Formal Environment Context Diagram

1.2.2.1 Import Facilities Several informal graphical models are widely used throughout the software industry for requirements analysis, specification, and even software design. Examples of such tools include Data Flow Diagrams, Structure Charts, SADT Diagrams, Concept Maps, Entity-Relationship Diagrams, and many others. In order to support the formal method approach to software engineering in a general sense, these informal models need to be transformed into the formal environment by automated means. One method of achieving this is to capture the essential information, associated with these abstract models of software requirements, and represent this information uniformly. The methods and procedures used to manipulate the uniform representation are contained within an abstract model manipulator illustrated in figure 1.5. The capture and transformation of essential data within these informal models is one subsystem of AFIT's formal environment.

1.2.2.2 General Transformation Facilities Information contained within the common abstract model must be transformed into a formal specification, or at least a template from which the formal specification can be completed. These transformations, contained within the general model transformer, become another

subsystem of AFIT's formal environment.

Since it is unlikely that the informal model contained all of the necessary information needed to build a formal specification, the transformation from the abstract model to the formal specification will most likely produce only a partial formal specification. The developer will then need to complete the formal specification using a formal specification environment, such as Refine^{TM}.

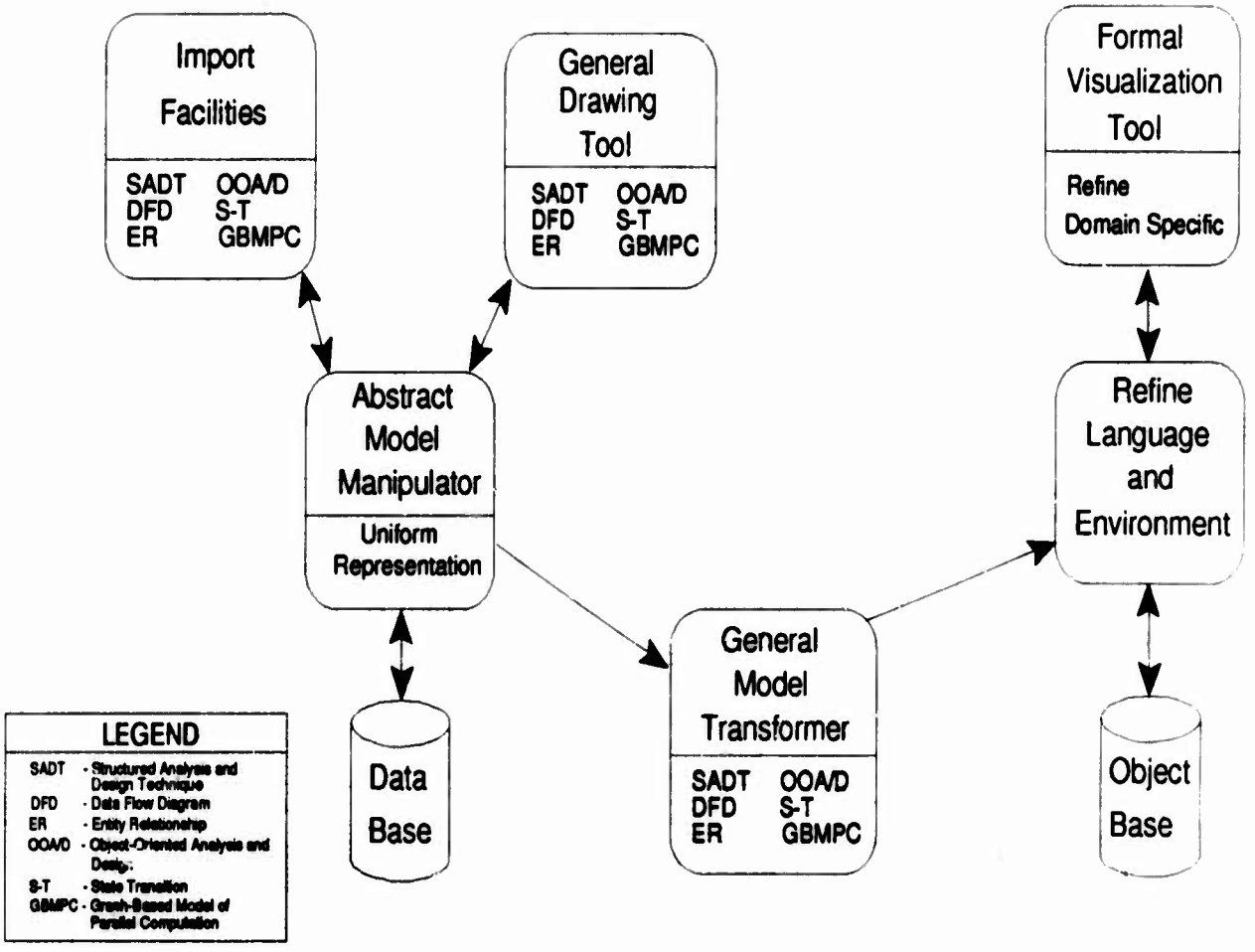


Figure 1.5. Major Sub-Systems of Formal Environment

1.2.2.3 Visualization Facilities The final sub-system of AFIT's formal environment is responsible for representing the formal specification via a formal graphical language. The development of this graphical language, along with a graph-

ical editor and bi-directional transformation models, is essential towards making the formal specification more understandable to current programmers, and non-technical users of the specification. The Refine formal specification language, developed and marketed by Reasoning Systems has been chosen for use within AFIT's formal environment. Through the use of the Refine Environment and associated object base (see Chapter II) partial and/or completed formal specifications are generated. These same partial and/or completed specifications are visualized through the use of the visualization facilities.

The primary purpose of the formal visualization tool is to enhance human (analyst and client) understanding of the formal specification. If this is to be achieved, the diagrams generated by the graphical language subsystem must meet the requirements for readability outlined in (16:18 - 19). Protsko et al. outline three main properties, which have been followed to the greatest extent possible, as follows (16:18 - 19):

1. Limited Complexity: Positioning of icons within the editor must insure sufficient white space to ensure readability.
2. The Concept of Flow: Positioning of icons within the editor must allow for the "flow" inherent with any specification. For example the begin symbol should be on the left or top of the diagram, while the stop symbol should be on the right or bottom respectively.
3. Placement Criteria: To enhance readability, the editor must follow certain placement criteria. For example, the diagram name should always be placed in the same location (absolute placement), or icons representing some result should always be to the left or below icons representing the arguments of the calculation (relative placement).

1.2.3 Objectives of This Thesis This research effort is primarily concerned with three main objectives:

1. Develop a formal graphical specification language which represents the Refine formal specification language. A Graphical notation for the Refine wide-spectrum formal specification language has been developed by Place in (15:5-1 - 5-34). This notation will form the basis of Visual Refine, a formal graphical language, and therefore, must be analyzed for the following:

   - adequate coverage of the Refine language, and
   - existence of a one-to-one mapping between Place's graphical notation and the Refine wide-spectrum specification language.

2. Develop and prototype a generalized mechanism for transforming the Refine wide-spectrum specification language into the Visual Refine graphical language. This objective is primarily concerned with applying the technology developed in objective number one. Once Visual Refine is well defined, it must be realized, along with an automated set of transformation tools, to be useful as a visualization tool for the Refine wide-spectrum specification language.

3. Extend the technology developed in objectives number one and two to any formal language that can be processed by the Refine object base. One example of this type of extension would be applying this visual technology to a Domain Specific Language used in a formalized Domain Specific Software Architecture.

The first objective is necessary to formalize Place's graphical notation, producing the Visual Refine formal graphical language. Objective number one is the foundation upon which the rest of this research is built. Objective number two demonstrates the usefulness of the previously developed Visual Refine language. Visualizing the complex structures, common to formal specifications, significantly reduces the complexity and enhances understanding, thereby making Balzer's transformational life cycle model more practical. Objective number three demonstrates the usefulness of formal language visualization towards enhancing understanding in domain specific environments. Together these objectives fulfill the requirements

of the formal visualization subsystem of AFIT's formal environment. Figure 1.6 illustrates the essential components of this thesis.

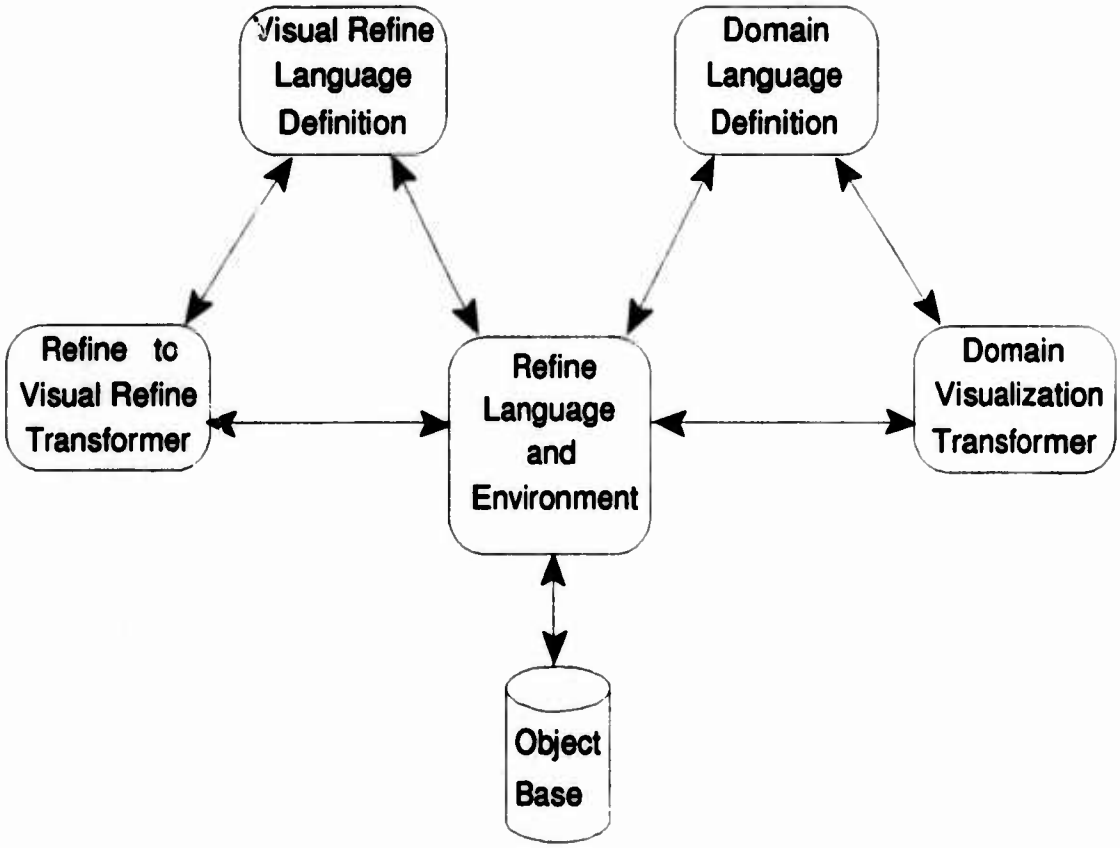


Figure 1.6. Formal Specification Visualization Tool

1.3 Summary of Current Knowledge

Research directed specifically at formal graphical specification languages is extremely rare, indicating the ground breaking nature of this type of work. Of numerous sources reviewed, only a few were found to contain information relevant to the visualization of formal specifications. Chapter II contains summaries of these sources. This research effort uses the Refine wide-spectrum specification system (19). Since this type of environment is fairly new, Chapter II also contains a synopsis of

the Refine system, as well as an overview of the important features used to achieve the objectives of this research.

1.4 Assumptions

No assumptions have been made in regards to previous research efforts or concurrent research efforts. The graphical notation presented in (15:5-1 – 5-62) forms the foundation of the formal graphical language developed; however, no assumptions have been made as to the completeness or correctness of the notation. A thorough analysis of Place's notation is conducted in Chapter III of this thesis.

1.5 Document Organization

Chapter II, Summary of Current Knowledge, presents a review of two foundational research efforts, (2) and (15), which provided the background needed to begin this research effort. Additionally, this chapter includes a synopsis of the Refine specification system (19), and an overview of important features used to formally specify the transformation systems developed for this research.

Chapter III, The Development of Visual Refine, systematically develops the Visual Refine graphical specification language. Beginning with Place's graphical notation for the Refine wide-spectrum specification language, each icon is formalized by first defining its input and output ports and then defining a one-to-one relationship between each icon and its Refine abstract syntax tree structure. Finally, a set of additional icons, necessary for Visual Refine to function properly, is developed.

Chapter IV, The Application of Visual Refine, demonstrates the usefulness of Visual Refine. This chapter develops a generalized method to transform Refine compiled source code into a pictorial representation in Visual Refine.

Chapter V, Domain Specific Languages, demonstrates the applicability of formal language visualization in the area of domain specific languages.

Chapter VI, Conclusions and Recommendations, summarizes the work accomplished by this research with specific conclusions as to the benefits and shortcomings. Recommendations for further research are also presented in this chapter.

Following the text of this thesis are several appendices. Appendices A through C are included to help the reader better understand how Place developed the graphical notation upon which Visual Refine is based. Appendices D and E are the results of the research work discussed in Chapter III. Appendix F is the Refine specification for the prototype of the Visual Refine transformation system. Appendices G and H are the Refine and Visual Refine specifications respectively for Place's solution to Kemmerer's library problem discussed in Chapter IV. As an alternative to Place's functional oriented solution, Appendices I and J provide the Refine and Visual Refine specification for Blankenship's object-oriented solution to the same problem.

II. Review of Current Literature

2.1 Introduction

Formal specifications are a fairly new idea within the fields of Computer Science and Software Engineering. The idea of visualizing a formal specification language is even newer. For this reason, very little has been written which directly relates to research in the area of visualization of formal specifications. There is however some preceding research which builds a foundation for the visualization of the Refine formal specification language. Bailor's research, "Theory for Graph-Based Language Specification, Analysis, and Mapping with Application to the Development of Parallel Software", provides a formal foundation for transforming between multi-dimensional languages (diagrams) and linear languages (high level programming language) (2). And Place's research, "The Development of a Graphical Notation for the Formal Specification of Software", provides a basis for the Visual Refine formal graphical specification language developed as part of this research (15). This chapter reviews the relevant portions of Bailor's and Place's work. Since wide-spectrum formal specification languages and environments are not commonly understood, this chapter will also include an overview of the Refine formal specification language and environment.

2.2 Theory for Graph-Based Language Specification, Analysis, and Mapping with Application to the Development of Parallel Software

The research work accomplished by Bailor in (2) is his attempt at, "providing the necessary formalism, theoretical foundations, and frameworks needed to develop *formal* specification diagrams for software systems" (2:ii). Bailor develops a formal language theory model of graphs and graph-based languages, which he calls a "graph generative system" (2:1-1). He uses this model as a basis for specifying, analyzing, mapping, and applying graph-based languages in the specification and design of par-

allel software systems (2:1-1), although the theory is applicable to specification and design of software systems in general. Three chapters of Bailors work are of particular interest to this review. Chapter VIII, "Graph- Based Language Mappings" and Chapter IX, "Existence and Analysis of Meta-Level Generating Functions" provide the mathematical foundation for graph-based language transformations, and Chapter X, "Application of the Theory", provides a generalized framework useful towards applying the previously developed theory. The frameworks developed by Bailor along with the supporting theory, form the necessary theoretical foundation from which the current research is derived. Following is a discussion of Bailor's frameworks and the supporting theory.

2.2.1 Application Frameworks Bailor develops both specific and general frameworks for applying his theory of "Graph-Based Generative Systems". Only the general frameworks are of interest to this review. First a general framework, applicable to the areas of visual programming, software development, and conceptual programming is presented (2:10-1 - 10-4). The primary benefit of formalizing and using graph-based languages for these types of systems is their ability to represent complex system, and software, relationships visually (2:10-1). It is possible that the formalized graph-based model is all that is required to adequately model the system. It is more likely though, that the formalized model must further be developed into a specification, design, and implementation, which will require various transformations from the original formalized graph-based model (2:10-1).

Figure 2.1 illustrates, in a general sense, how graph-based languages can be used to represent several other formalized languages (2:10-2(figure 10.1)). By using the meta-level graph generative systems discussed in section 2.2.2, the graph-based languages can be formalized sufficiently enough to allow a parser and compiler to be constructed which will recognize graphs from the specified class and transform these graphs into the desired specification language (2:10-1 - 10-3). The transformations of interest to this review are the transformations from the graph-based language into

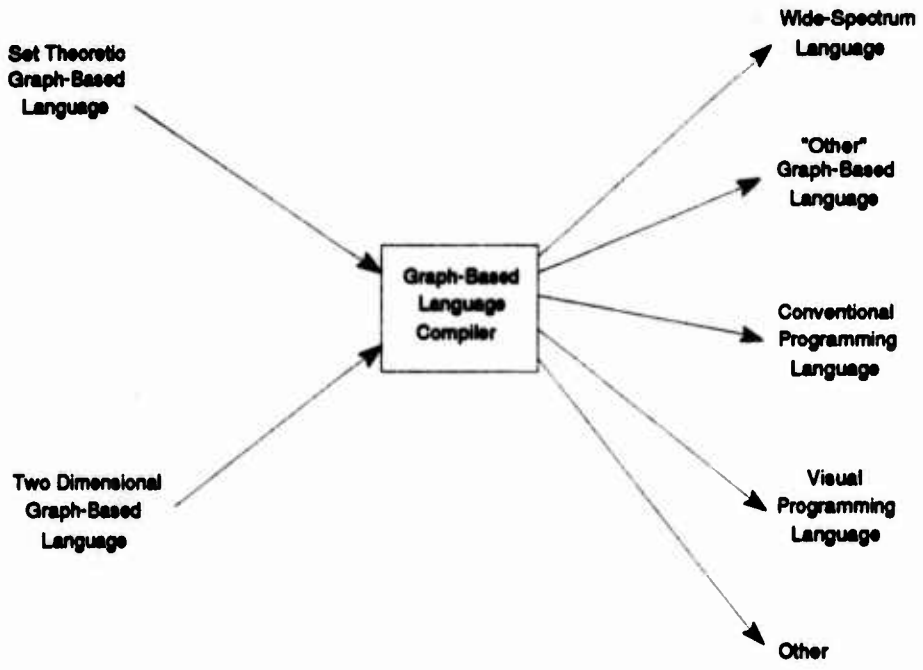


Figure 2.1. General Application Framework

either a wide-spectrum language or a visual programming language. By using the relatively easy to use and understand graph-based language some of the difficulty in using and understanding wide-spectrum languages can be alleviated (2:10-1). When addressing the transformation to visual programming languages, Bailor states that this should be a natural application of the theory; specifically, given sufficient detail about the syntactical constructs of the graph-based language, both the domain of interest and the software specification and design can be modeled (2:10-3). He refers to this as a wide-spectrum, graph-based language (2:10-3). It is also true that given sufficient detail about the software specification, a compiler or translator can be developed which generates a graph-based language. It is this inverse approach which is important to the development of Visual Refine.

Next Bailor provides a framework to support the program transformation, or automation-based, paradigm (2:10-4 – 10-5). Figure 2.2 provides an illustration of this framework (2:10-4 (figure 10.2)). Using this framework, the formalized graph-

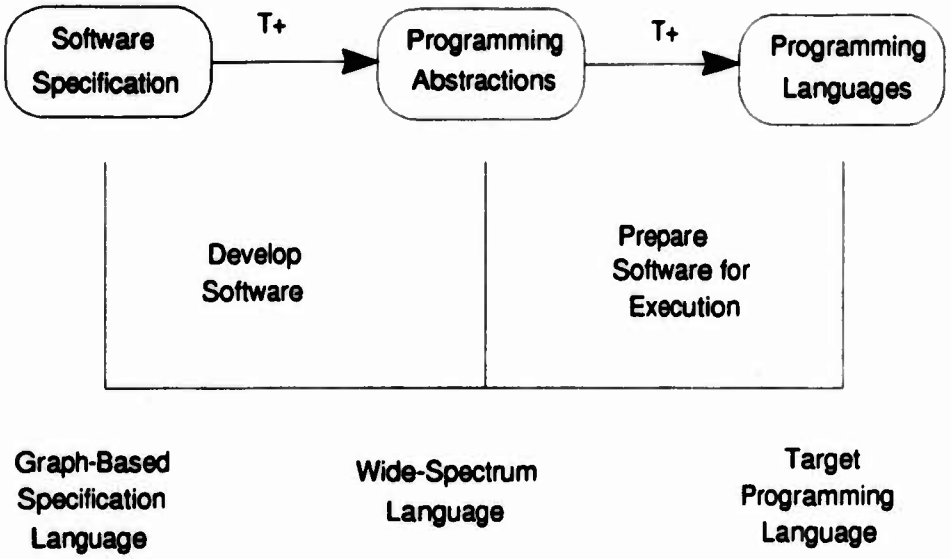


Figure 2.2. General Framework for Software Development

based language is used to initially specify the software system. For example, a finite state machine could be specified as a directed graph using the graph-based language. Then through a series of transformations, the graph-based language is converted to a wide-spectrum language and finally into the target programming language (2:10-4).

The graph-based language can also be used to visualize, in a conceptually easier manner, the more complex syntactic structures of the wide-spectrum language, thus making wide-spectrum languages more understandable to the average user. Bailor points out that the syntactical structures of the formalized graph-based language must be mapped to the syntactical structures of the wide-spectrum language, but in order to specify these mappings at the conceptual, mathematical, and implementation levels, an extensive analysis must be done (2:10-4). He further defines the following as the minimum set of mappings which must be defined (2:10-5):

1. Graph-Based Syntax to Linear Syntax — obvious mapping required for the formalized graph-based language to function as a front-end to current wide-

spectrum languages.

2. Linear Syntax to Graph-Based Syntax — mapping required to allow the formalized graph-based language to function as a mechanism for visualizing the wide-spectrum language.
3. Graph-Based Syntax to Graph-Based Syntax — allows for varying levels of abstraction within the formalized graph-based language.
4. Linear Syntax to Linear Syntax — allows for mappings between components which are not necessarily multidimensional.

It is precisely these types of mappings that the current research is interested in showing. In order to develop a visual representation for either a formal specification language, or a domain specific language, mappings number one and two above must be determined.

2.2.2 Supporting Theory Bailor defines three levels of detail at which language mappings can be discussed and four basic classes of languages which are of interest to his research. The levels of detail are (2:8-1 - 8-2):

- Conceptual Level — an informal level where basic objects/entities within languages are mapped one to another.
- Mathematical Level — a formal level where a mathematical function, or relation, is used to map a precisely defined domain into a precisely defined range.
- Implementation Level — a formal level, based on the mathematical level, where additional details are included such that algorithms and their associated data structures can be used to implement the mapping.

Although the conceptual level of abstraction may be used for describing how a specific mapping functions, it is the implementation level of abstraction which is of most importance to this review. Of the four basic language classes which Bailor describes, only two are of importance for this review; they are (2:8-2):

- Two (Multi) Dimensional, Graph-Based Languages — Two, or multi, dimensional representation (a diagram) which implicitly represents the sets used to describe graphs.
- Linear (or one dimensional) Languages — used in graph-based language compiler systems as possible target languages. Examples include software specification languages and programming languages.

Of particular interest is Bailor's use of language translation functions (2:8-5) to specify translations between two dimensional languages and linear languages. Conceptually a language translation function is a device, called a translator, which takes as input an element from the source language and produces an output which is an element in the target language (2:8-7). Desirable features of such a device are (2:8-7):

- Each input/output pair within the translation definition should be easily determined.
- Given a translation definition, it should be possible to mechanically construct a translator with the following features:
  1. Efficient in terms of both time and space requirements,
  2. Small size, and
  3. Correctness.

Next Bailor presents "Generalized Translation Functions for Graph-Based Languages" (2:8-8 – 8-12). Several functions are presented, but for our purposes only the translations between two dimensional languages and linear languages are of concern. There are two translations of interest; they are (2:8-11 – 8-12):

$$f : L_G^{2D} \rightarrow L_L$$

meaning a function (f) such that a 2-Dimensional Graph-Based input language (L_G^{2D}) is mapped to a Linear output language (L_L) and

$$f : L_L \rightarrow L_G^{2D}$$

meaning a function (f) such that a Linear input language (L_L) is mapped to a 2-Dimensional Graph-Based output language (L_G^{2D}). For a more complete description of the notation used by Bailor see (2:8-8 – 8-9). To further define the above translations, the syntactic structures contained within the Linear language must be explicitly defined (2:8-12).

2.2.3 Existence and Analysis of Meta-Level Generating Functions A meta-level language is simply a language which describes another language, and in a sense, they define a hierarchical structure for languages themselves (2:9-1). Bailor demonstrates the existence of meta-level generation functions for graph generative systems (2:9-1). A minimal set of rules exists within a meta-level language, providing the ability to specify a much larger set of graphs. Specifically, the derivation of even complex graphs can be carried out in a straight forward and uncomplicated manner by applying only a relatively small number of rules (2:9-1). This type of language provides the user with an easy environment to work in, but unfortunately this ease of use is usually at the expense of efficiency (2:9-1). According to Bailor, the benefit of a meta-level generation system is that "it provides a computational process that can be analyzed" (2:9-1). Particularly, various complexity metrics can be applied, and relationships between various processes on the graph can be determined.

Bailor shows the existence of two specific meta-level graph generative systems. They are (2:9-5 – 9-22):

1. Set Theoretic Systems, and
2. Two Dimensional Systems.

These two systems are shown to be equivalent. The mathematical details concerning these generative systems are found in (2:9-2 - 9-24); however, analysis of these generative systems shows that,

> [these] meta-level systems can generate all graphs with arbitrary but finite node and edge sets, all graphs with arbitrary but finite degree, all graphs that contain arbitrary but finite length open and closed walks, and all graphs containing an arbitrary but finite number of connected components (2:9-25)

Since all diagrams derived using the Visual Refine language, explained further in Chapter III, can be defined as two dimensional graphs, the meta-level generative functions developed by Bailor directly apply.

2.3 The Development of a Graphical Notation for the Formal Specification of Software

Place's research develops a graphical formal specification language based on the Refine wide spectrum language. Refine formal specifications are represented in a graph-based iconic manner, making the graphical language much easier to create and manipulate than the equivalent textual formal specifications (15:xiii). Place's work is centered around the formal transformation life cycle model developed by Balzer in (5). Place's work is intended to make formal specifications more easily understood, by visualizing the specification, via an iconic-based language. If formal specification can be made understandable to the average user, then the formal transformation life cycle model could become the paradigm of choice for software development in the future.

If the formal transformation life cycle model is to be used, Place believes that the difficulties associated with understanding formal specifications, the foundation of the model, must be resolved (15:6). Since humans tend to understand diagrammatic representations easier than textual representation, Place concentrated his efforts on

the development of a graphical notation for the Refine wide-spectrum formal specification language in support of the formal transformation life cycle model (15). Of importance to this review is Chapter five of Place's thesis, "A Graphical Representation for the Refine Specification Language" (15).

In order to represent the Refine language, Place first decomposed the language into a set of primitives (15:5-2 - 5-19). The implicit parse tree contained in the Refine language documentation is used by Place to guide the decomposition (15:5-2). Place identifies Refine's primitive data types and then identifies Refine's primitive operations, categorizing the operations by their operand data types. He also provides a categorization by operation characteristics. The results of Place's decomposition are contained in Appendices A and B.

Once the decomposition was done, a graphical notation for Refine was developed (15:5-16 - 5-35). Place uses rectangular shapes to represent program data, circles or ellipses to represent program operations, solid arcs to indicate data flow, and dashed arcs to indicate control flow (15:5-16 - 5- 17). He then systematically constructs icons (15:5-19 - 5-35). For a complete listing of Place's graphical notation see Appendix C. It is important to note, while reviewing Appendix C, that many of the graphical symbols are used for more than one operation. These "overloaded" symbols perform intuitively similar functions on dissimilar objects (15:5-18). If this notation is to be used, automated tools must be able to recognize various Refine structures for each overloaded Visual Refine symbol.

The syntax of the graphical language is presented in general terms as follows (15:5-34):

> an operator node must have a least one data flow input into the operator with any number of additional data and control flow inputs, and that an operator must have at least one data flow output and any number of additional data and control flow outputs

Specific syntax, meaning restrictions on the number of inputs and/or outputs, is governed by individual operators themselves.

2.4 The Refine Formal Specification Environment

Refine^{TM} is a wide-spectrum formal specification language developed and marketed by Reasoning Systems. As a wide-spectrum language, Refine is applicable over several areas of software development. Refine is particularly useful for modeling system behavior as part of the requirement analysis, or domain modeling functions. However, because Refine is an executable formal specification, once the systems behavior is determined, the resulting formal specification easily serves as a mechanism for developing the software design. The following three sections describe three of the key components of the Refine Development Environment, used to prototype the technology developed as part of this research. Figure 2.3 shows the relationship between each of the following Refine components.

2.4.1 Refine Language and Object Base Refine consists of two main components, the Refine object base and the Refine specification language (19:1-2). The Refine Object Base maintains a vast amount of information about the system being developed, including detailed information about the specifications themselves. The Refine specification language, however, is used to manipulate the object base, and by so doing, models a systems behavior. The Refine specification language is lexically analyzed and parsed into an abstract syntax tree (AST). This AST is represented in the Refine object base with each node of the AST being represented as an object and each arc of the AST being represented as an attribute of the object (19:5-4 – 5-22). Refine provides several facilities for recognizing and manipulating objects. These facilities allow the Refine environment to transform the formal specification from high level requirements into detailed software specifications, and even to implementation if desired. The Refine object base also facilitates the transformations discussed above, for translating linear, or textual, syntax to the two-dimensional,

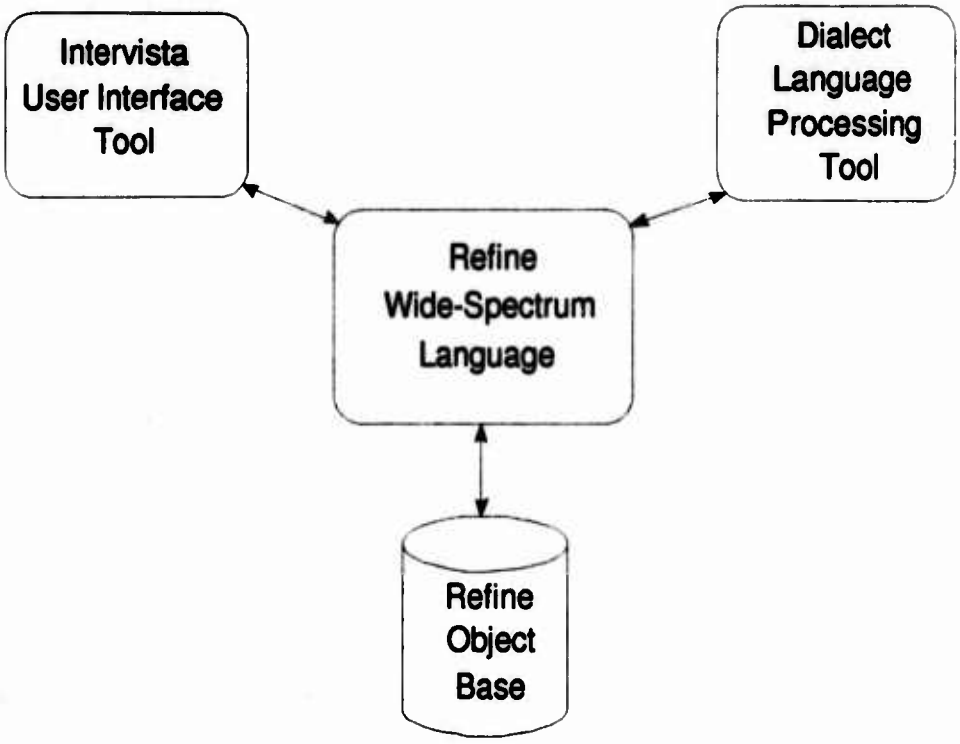


Figure 2.3. Interrelationship Between Refine Components

graph based syntax of Visual Refine.

2.4.2 Intervista Intervista is a Refine subsystem, loaded separately into the Refine object base, used to create interactive graphical and textual user interfaces (18:1-1). Although not overly sophisticated, Intervista provides a capable graph-based diagraming facility. Four types of predefined icons are available (18:7-8) and they are:

1. Box,
2. Diamond,
3. Ellipse, and
4. Text.

Both dashed and solid arcs are supported in Intervista. A wide range of predefined functions are also available for manipulating windows (18:7-6 – 7-8), surfaces (18:7-5,

7-6), icons (18:7-8 - 7-10), and links (18:7-10 - 7-13). Intervista easily provides the facilities required to prototype the visualization technology of this research.

2.4.3 Dialect Dialect is a Refine subsystem, loaded separately into the Refine object base, used to define and manipulate formal languages. Formal languages are input using a BNF type high-level syntax description language. From this language description, an LALR(1) parser, pretty printer, and pattern matching facilities are automatically produced (17:1-2, 1-3). Dialect also inserts all code necessary to parse the language into an abstract syntax tree representation within the Refine object base. These powerful language processing facilities are essential to providing an effective domain specific modeling and visualization environment.

2.5 Conclusion

Bailor's work has provided a solid theoretical foundation useful for developing formalized graph-based languages, such as Visual Refine. He has also provided the necessary frameworks for applying this theory in the area of software specification and design; the actual transformation methods, which are needed for these frameworks, are the primary subject of the present research. Place's work has resulted in a set of icons suitable for describing the Refine formal specification language. By using the theory developed by Bailor and the icons developed by Place, a formalized graph-based language, based upon the Refine wide-spectrum language, can be developed and is presented in Chapter III. This formalized graph-based language, called Visual Refine, can then be used in support of Balzer's automation based paradigm. Graph-Based languages are conceptually easier to use and understand; Wide-Spectrum languages are powerful but conceptually difficult to understand. By developing formal transformations between Refine and Visual Refine the difficulty of understanding wide-spectrum formal specification languages can be significantly reduced.

III. The Development of Visual Refine

3.1 Introduction

Place's research, "The Development of a Graphical Notation for the Formal Specification of Software", develops a graphical notation which represents the Refine wide-spectrum formal specification language (15:Chapter 5). Place states that a one-to-one and onto translation scheme exists between expressions using his graphical notation and corresponding Refine expressions (15:5-35). Furthermore, he asserts that after accounting for the overloading of certain graphical operators, a one-to-one correspondence exists between operators of the two languages (15:5-35). Although these statements may in fact be true, there is nothing inherent in Place's development methodology, nor does he provide any supporting evidence, to adequately support such assertions. In order for Place's graphical notation for the Refine wide-spectrum specification language to be used as the foundation for the Visual Refine graphical specification language, it must first be shown that the above one-to-one relationship exists. In the few cases were ambiguity is found, Place's notation is modified to eliminate the ambiguity. Additionally, Place's notation is extended to provide essential Refine functionality not originally accounted for in his notation. The results of this evaluation and modification of Place's notation is a formal definition of the Visual Refine graphical specification language.

This chapter is divided into three sections. Section 3.2 explains the development methodology used to construct the Visual Refine graphical specification language from Place's Refine graphical notation. Section 3.3 contains a detailed example of how the methodology is applied, using selected examples from various stages in the development process. And finally, section 3.4 formally defines the one-to-one relationship established between the Visual Refine graphical specification language and the Refine wide-spectrum specification language.

3.2 Development Methodology

One of two possible approaches can be used to construct a visual representation of the Refine language. Using the first approach, a one-to-one relationship is developed between the Visual Refine graphical notation and the explicit Refine text, referred to in (19) as the surface syntax. An example of this type of mapping might be

where *structure* is an expression evaluating to a map, set, or sequence. To implement this type of approach the Visual Refine language would need to be rigorously defined, with an explicit grammar and set of production rules. A compiler would then need to be developed to transform from the Refine surface syntax into the Visual Refine graphical representation. A separate compiler is needed for transforming from Visual Refine into Refine surface syntax.

Using the second approach, a one-to-one relationship is developed between the Visual Refine graphical notation and an alternative intermediate representation of the Refine wide-spectrum specification language. A Refine program is represented as an abstract syntax tree (AST) within the Refine internal object base. A Visual Refine program can also be represented as an AST; thus, making the Refine object base representation of an AST a good intermediate representation for both languages. Because the Refine environment, and accompanying object base, can readily express both Refine programs (19:5-4 - 5-22) and Visual Refine syntax (18), the second

approach has been chosen for use by this research effort. Development of the Visual Refine graphical specification language is explained further in this chapter with this approach in mind.

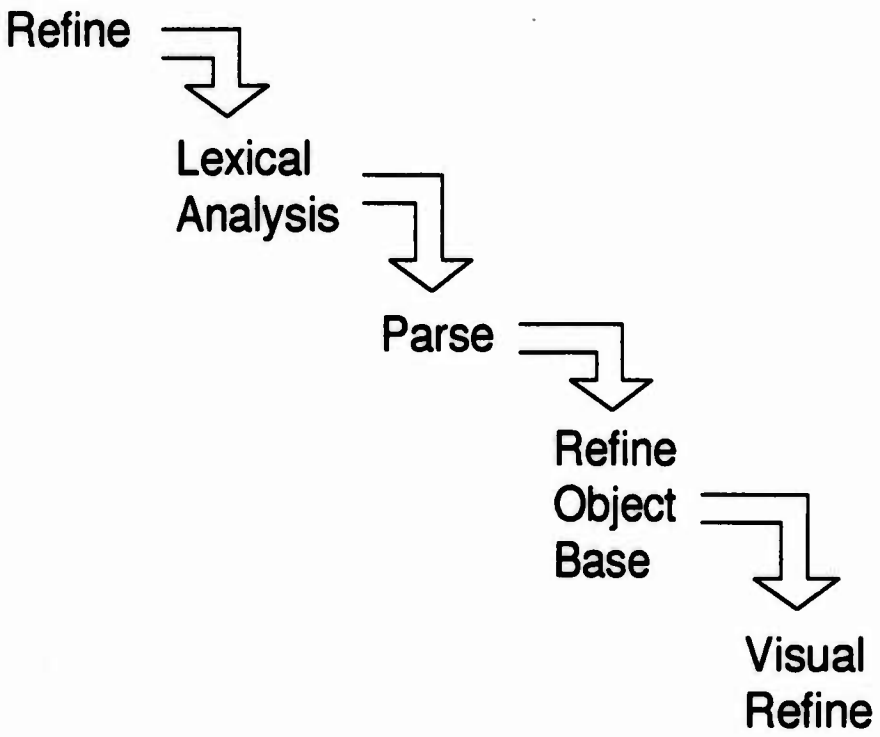


Figure 3.1. Refine to Visual Refine Transformation

Figure 3.1 illustrates where Visual Refine fits into the transformational process. The greatest benefit of Visual Refine is derived by making the difficult to understand structures of the Refine wide-spectrum language more easily understood through visualization. Therefore, the visualization of the object base becomes a natural representation of any Refine syntactical structure. The process of transforming Refine syntax into the Refine object base is a function of the Refine compiler, and therefore not discussed further in this thesis. However, the object base and the Refine language representation in the object base are of further interest.

Refine represents syntactical structure in the object base as an AST, with each node of the AST being represented as an object and each arc in the AST represented as an attribute of an object (19:5-4 – 5-8). The Refine environment allows for general

access to this object base, and even provides several functions for manipulating and processing the AST structures within the object base (19:3-186 - 3-197). Since the Refine environment allows access to the parsed representation of the Refine language, and it is this representation which Visual Refine is representing, the production rules for Visual Refine are implicitly defined as being equivalent to the production rules of the Refine wide-spectrum language. Using this knowledge, Visual Refine is defined by showing a one-to-one relationship between icons in Visual Refine and objects in the Refine object base.

The following is a concise set of steps used to develop the Visual Refine graphical specification language:

1. Begin with a clear description of Place's graphical notation for the Refine wide-spectrum specification language.
2. For each of Place's icons, show the Refine object base AST structure for which the icon represents.
3. Formalize Place's icon to include definitions of input and output ports to each icon.
4. Define any additional icons necessary for general applicability.
5. Define Visual Refine icons such that each icon, including the input and output arcs, maps to one and only one Refine object base AST structure.

The first three steps are further explained in section 3.3. The last two steps are developed in further detail in section 3.4.

3.3 Detailed Example

Place's graphical notation for the Refine wide-spectrum specification language provides icons for a broad range of Refine operations. Yet Place's notation lacks the formalism necessary to implement his notation in a meaningful way. This section

provides several examples to illustrate the process used to formalize Place's graphical notation. This formal definition of each icon is used in development of the Visual Refine graphical specification language developed in section 3.4.

3.3.1 Place's Graphical Notation The first step towards defining Visual Refine is to list the icons developed by Place (15:5-16 - 5-33). A complete listing is contained in Appendix C. Contained in this section is a partial listing used to illustrate the process followed to formalize Place's notation. Figure 3.2 illustrates the

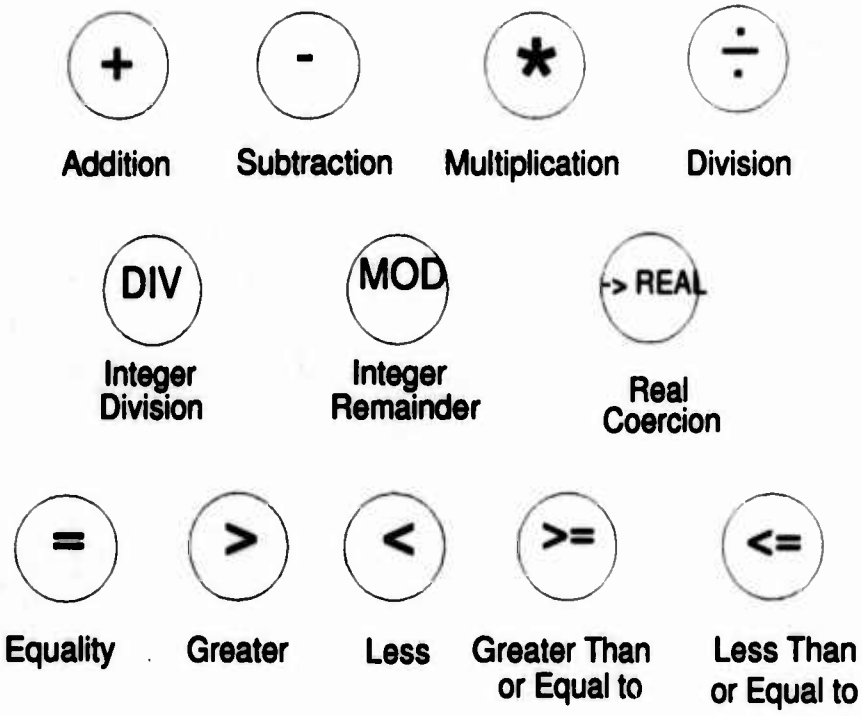


Figure 3.2. Mathematical Operations

mathematical operation icons developed by Place (15:5-20). These icons have been chosen for the example because they are conceptually easy to understand and have universally accepted notations. Figure 3.3 illustrates the set operation icons developed by Place (15:5-24). The set operation icons have been chosen to illustrate how the overloading of the plus (+) and minus (-) icons is handled, as well as to show the resolution of non-unique AST structures discussed later.

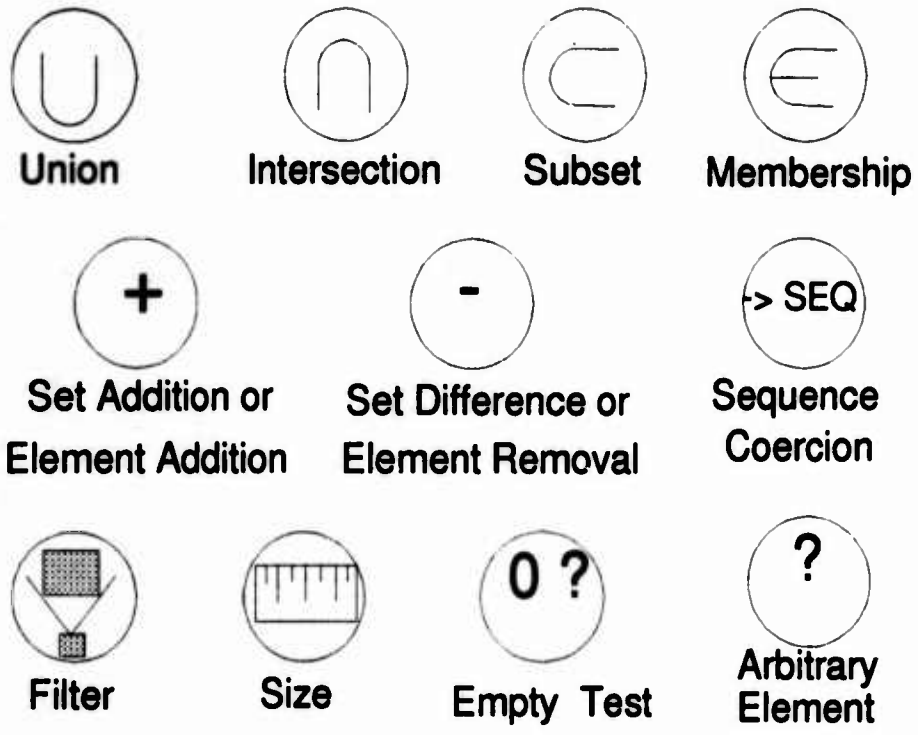


Figure 3.3. Set Operations

3.3.2 Refine Abstract Syntax The next step is to determine what the abstract syntax structure looks like for each icon Place has defined. To accomplish this step, Place's decomposition of the Refine wide-spectrum specification language (15:Appendix D) and the Refine users manual (19:3-20 – 3-150) had to be consulted. From this information, a determination is made as to the Refine surface syntax intended by Place in his notation. Using the Refine command interface (19:4-3 – 4-6), the surface syntax previously determined is parsed, resulting in the creation of an abstract syntax tree representation defined as the current object of the Refine object base. To display an object, along with its attributes, the Refine "(pup)" command (19:4-30 – 4-32) is used. By using the "(pup)" command, along with commands for moving within the object base (19:4-57 – 4-61), the abstract syntax tree structure is determined. Although the above procedure may sound complicated, the Refine environment makes it a fairly straightforward process. Appendix D contains the abstract syntax tree structures, derived using the above methodology, for each of

the graphical icons developed by Place and listed in Appendix C.

To further illustrate the process used, the multiplication mathematical operator and the subset set operator, shown in figure 3.4, will be developed in detail. Multiplication is listed under 'Numbers' in Place's decomposition of the Refine language by operand data type (Appendix A), and is not shown to be overloaded in his decomposition by operation characteristics (Appendix B). Therefore, the 'Numbers' section of the Refine Users Manual (19:3-20 - 3-35) is consulted to determine the intended Refine surface syntax for the multiplication operation. Similarly, the sub-

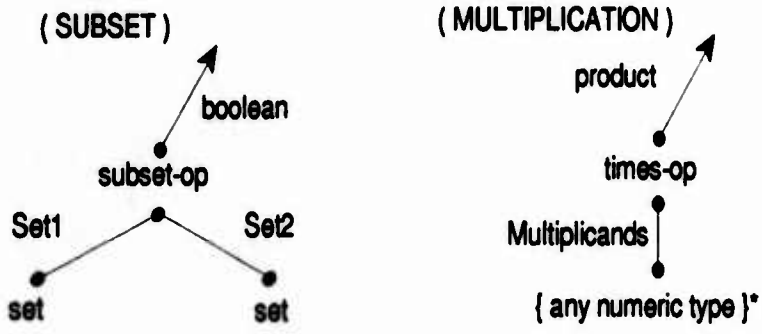


Figure 3.4. Multiplication and Subset Operations

set operator is listed under 'sets' in Appendix A, and not shown to be overloaded in Appendix B. The 'Sets' section (19:3-56 - 3-77) is therefore consulted for the subset operation. The next step is to determine the most appropriate surface syntax. In the case of multiplication, the most generic, and least restrictive, surface syntax is an asterisk (*) followed by a left parenthesis, zero or more numeric items, and finally a right parenthesis (19:3-26). The surface syntax for the subset operation is always a set, followed by the word "subset", followed by another set of the same type (19:3-76.

3-77). The next step uses the Refine command interface to determine the abstract syntax tree structure.

To determine the multiplication AST structure, the command "(#> *())", representative of a multiplication operation, is input to the Refine command interface. This results in an AST being built and represented as the current object in the Refine environment. The "(pup)" command (19:4-30 - 4-32) is used to display the current object, which has the following form:

```
a re::times-op
   re::class:  re::times-op
   re::multiplicands: {}
   re::top-object-read: true
```

Similarly, to determine the subset AST, the command "(#> {} subset {})", representative of a subset operation, is input to the Refine command interface and the resulting object, when displayed, appears as follows:

```
a re::subset-op
   re::class:  re::subset-op
   re::set2: #1<a re::literalset-op>
   re::set1: #2<a re::literalset-op>
   re::top-object-read: true
```

The final step is to analyze the object to determine the AST structure. An analysis of the above objects easily produces the multiplication and subset abstract syntax tree structures depicted in figures 3.5 and 3.6. Input data type, and resultant information, is determined by reviewing the operational behavior descriptions contained in (19:3-26, 3-27, 3-76, 3-77).

The same methodology is used to develop AST structures for each icon listed in Appendix C. Figures 3.5 and 3.6 illustrate the AST structures for the icons shown in figures 3.2 and 3.3 respectively. The plus (+), and minus (-) set operations icons

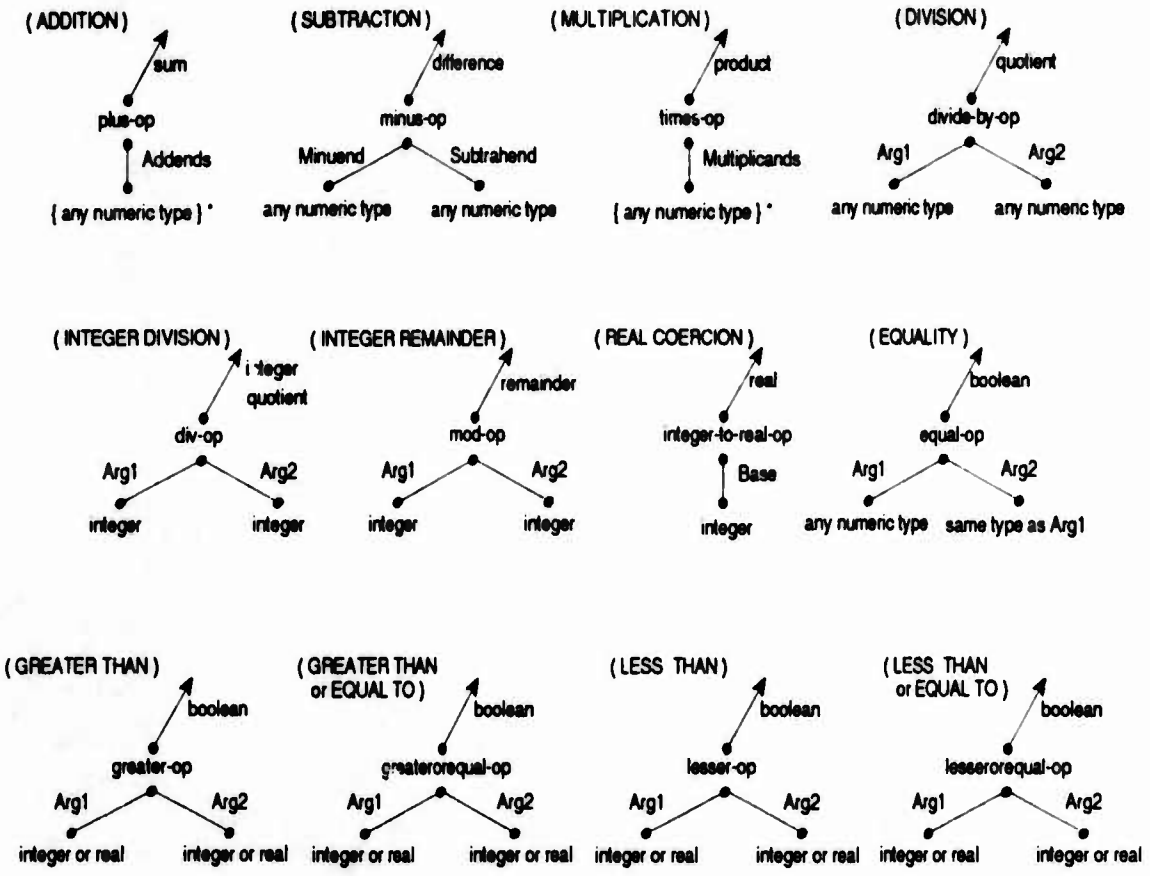


Figure 3.5. Mathematical Operations

each require two AST structures to adequately represent the intended surface syntax, denoted by the dashed boxes in figure 3.6. These same icons are overloaded with the mathematical plus (+) and minus (-) icons, yet have significantly different AST structures. A resolution for both of these problems is discussed in section 3.3.4. A complete listing of AST structures for icons listed in Appendix C can be found in Appendix D.

3.3.3 Abstract Syntax Notation The notation used in figures 3.5 and 3.6, as well as in Appendices D and E, is taken from the Refine users manual (19) and is summarized as follows:

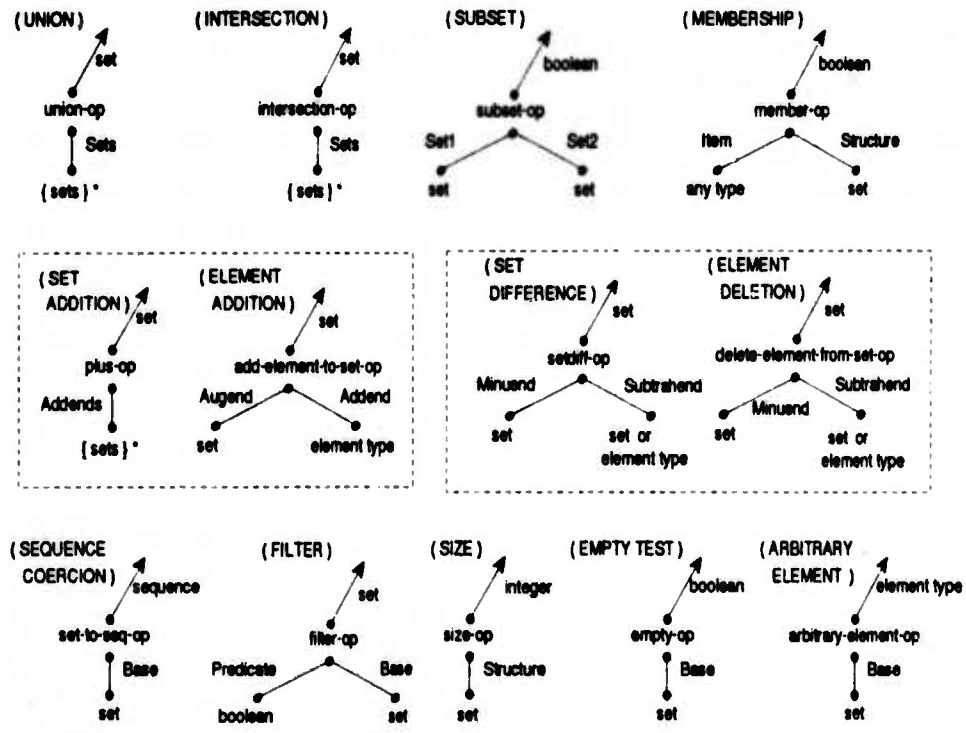


Figure 3.6. Set Operations

- an asterisk (*) means zero or more items
- a single plus (+) means at least one, possibly more, items
- two pluses (++) mean at least two, possibly more, items
- one or more items enclosed in square brackets ([*item*]) is a sequence
- one or more items enclosed in curly brackets ({ *item* }) is a set
- a vertical bar (|) is a logical OR
- an ampersand (&) is a logical AND

3.3.4 Formalized Graphical Notation Enough information is now available to formally specify the input and output ports for each of Place's icons. For most of Place's icons, this process is a matter of simply overlaying the AST structure with the graphical notation. For example, the multiplication AST structure, presented in figure 3.5, can have zero or more inputs to the 'times-op' node, denoted by the set

containing zero or more elements. The 'times-op' node produces exactly one output, denoted by the single arrow, which is the product of all inputs. This same AST structure is represented graphically in figure 3.7 by showing an input arrow with an asterisk (*), meaning zero or more inputs are possible, and a single output arrow, meaning one and only one output is produced. Similarly, the subset AST structure,

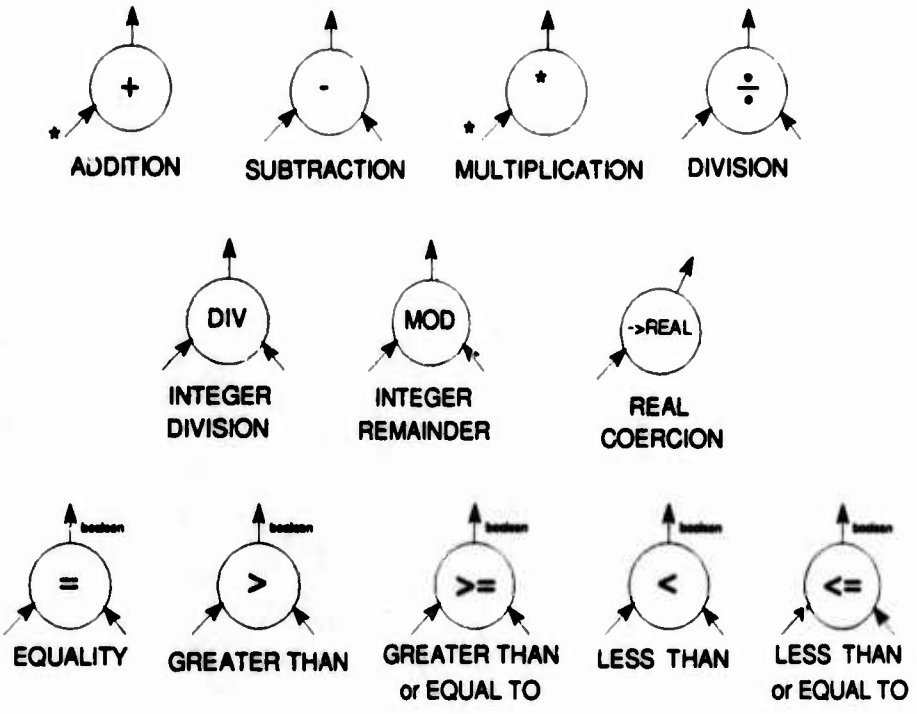


Figure 3.7. Mathematical Operations

presented in figure 3.6, has exactly two inputs, denoted by 'Set1' and 'Set2', and produces exactly one output, of type boolean, to indicate whether 'Set1' is a subset of 'Set2'. The graphical representation of subset, shown in figure 3.8, depicts this same relationship by illustrating exactly two input arcs and exactly one output arc for the subset icon.

Unfortunately, not all of the icons were as straightforward as these examples. The plus (+) and minus (-) set operations were viewed by Place as a case of overloading the mathematical plus (+) and minus (-) operations. This view turns out to be incorrect. The plus (+) set operation can have either a "plus-op" AST rep-

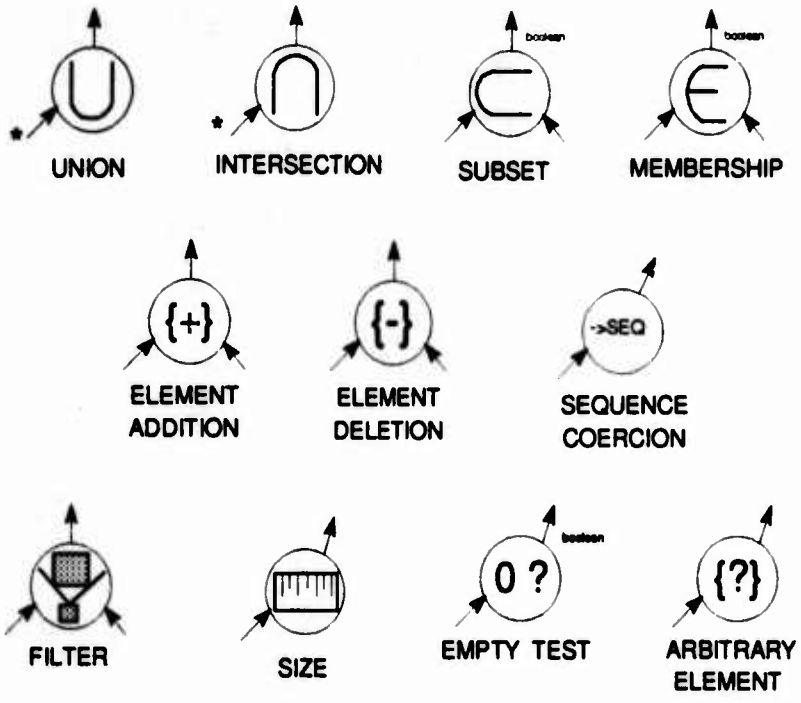


Figure 3.8. Set Operations

resentation or an "add-element-to-set-op" AST representation, as indicated by the dashed box in figure 3.6. If the "plus-op" AST representation is chosen as the underlying structure for Visual Refine then an overloaded graphical symbol can be used; however, this choice makes element addition to a set no longer possible. If the alternative "add-element-to-set-op" AST representation is chosen as the underlying structure for Visual Refine then using an overloaded icon produces an ambiguity in the underlying structure of this icon. Functionally, the addition of sets, using a "plus-op" AST structure, and the union of sets, using the "union-op" AST structure, is the same. For this reason, the inclusion of a set addition icon, based upon the "plus-op" AST structure, would not add any additional capabilities to the Visual Refine language. Therefore, the "add-element-to-set-op" AST structure is chosen to represent set addition. By making this choice, an ambiguity now exists in the underlying AST structure for the plus (+) icon. To resolve this ambiguity, a new icon is developed, and shown in figure 3.8, which represents element addition to a

set. This new icon represents the "add-element-to-set-op" AST structure.

The minus (-) set operation contains the same ambiguity problem found with the plus (+) set operation. In this case, the dashed box in figure 3.6 shows two similar AST structures, set difference ("setdiff-op" AST) and element deletion ("delete-element-from-set-op" AST), for representing the surface syntax intended by Place. Both of these structures are ambiguous with the mathematical minus (-) operation. A choice must be made as to which AST structure represents set subtraction. To maintain consistency with the plus set operation, the "delete-element-from-set-op" AST structure is used for Visual Refine. Again, making this choice , introduces ambiguity in the underlying AST structure for the minus (-) icon. To resolve this ambiguity, a new icon is developed which represents element deletion from a set. This new icon, shown in figure 3.8, has the "delete-element-from-set-op" as its underlying AST structure.

Additionally, figure 3.8 shows a notational change to the arbitrary element icon. Set brackets have been placed around the question mark (?) which represents the arbitrary element operation. This modification makes the arbitrary element icon more easily recognized as a set operation.

3.4 Formal Definition of Visual Refine

Finally, a formal definition of the Visual Refine graphical specification language can be made. Three classes of icons are discussed:

1. Non-overloaded icons which have a unique AST structure.
2. Icons which are overloaded, but easily resolved due to similar AST structures.
3. Additional icons developed for the Visual Refine graphical specification language.

The notation used for defining the Visual Refine graphical specification language is presented in the next section, followed by a more detailed discussion of each class of icons mentioned above.

3.4.1 Notation for Defining Visual Refine Visual Refine is defined by showing a one-to-one relationship between the icons of Visual Refine, and the AST structures of the Refine wide-spectrum specification language. To illustrate this relationship the following notation is used. Each Visual Refine icon, including input and output arcs, is listed. To the right of each icon is a double headed arrow, used to symbolize the one-to-one relationship, and to the right of the arrow is the Refine object base, abstract syntax tree structure, associated with the icon. Figure 3.9 shows pictorially the notation described above. Additional notational conventions are the same as

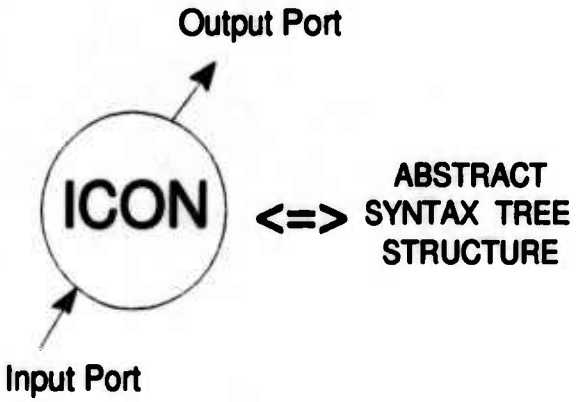
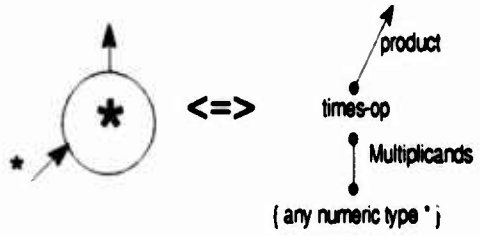
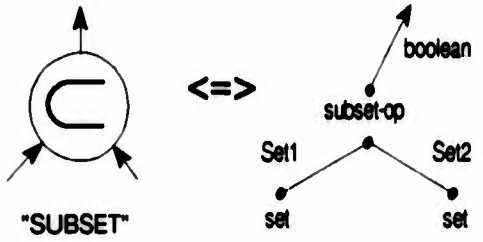


Figure 3.9. Notational Definition for Visual Refine

discussed earlier in section 3.3.3. The combination of the icon and the object base AST structure form the formal definition for the Visual Refine graphical specification language. The notation described above is followed in the remaining portions of this chapter, as well as in Appendix E, which contains a complete listing of the Visual Refine graphical specification language.

3.4.2 Non-Overloaded Icons The first icons defined are those icons which are unique in Place's notation, and also have a unique AST structure in Refine. Using the notation described above, these icons are mapped directly to the unique AST structure generated in section 3.3.2. For example, the multiplication and subset operators described in section 3.3.4, are formally defined as follows:





The first section in Appendix E contains the definitions of all Visual Refine icons which have this direct one-to-one relationship.

3.4.3 Overloaded Icons Next, all non-unique, or overloaded, icons developed for Visual Refine are defined. These icons require a further evaluation of the AST structures to determine what, if any, modifications are required in order to handle the overloading. All except the plus (+) and minus (-) icons, discussed in section 3.3.4, required only the combining of data types in the AST structure. For example the 'Size' Visual Refine icon can refer to the size of a map, the size of a set, or the size of a sequence, as shown in figure 3.10. The Visual Refine formal definition of this icon is obtained by combining the map, set and sequence data types with a logical

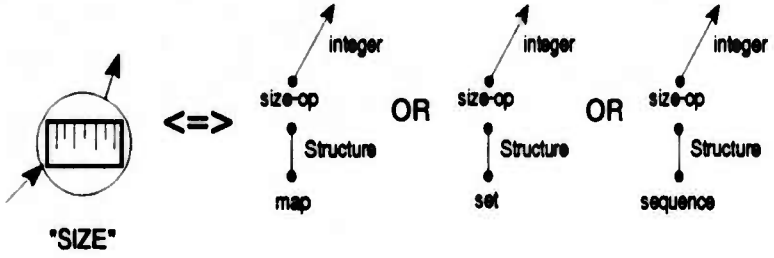
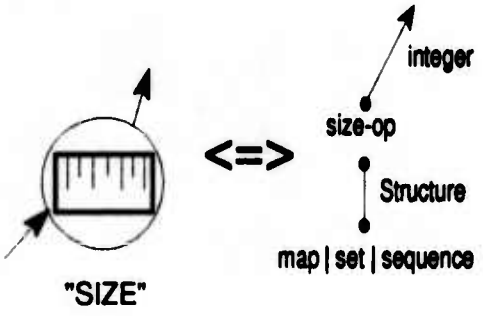


Figure 3.10. Overloading of the Size Icon

OR operator as shown in the following formal definition for the 'Size' operator:



The second section in Appendix E contains the formal definitions for overloaded icons. Each of these icons was developed using the same method as described for the 'Size' icon.

3.4.4 Additional icons The third and final class of icons developed for Visual Refine are necessary in order to provide a sufficient set of icons for general use. The formalization of Place's graphical notation has resulted in six additional icons being developed. Additionally, definitions for literal integers and strings have been developed. Figure 3.11 illustrates each newly developed icon, along with its object

base abstract syntax tree structure. The need to develop the element addition and

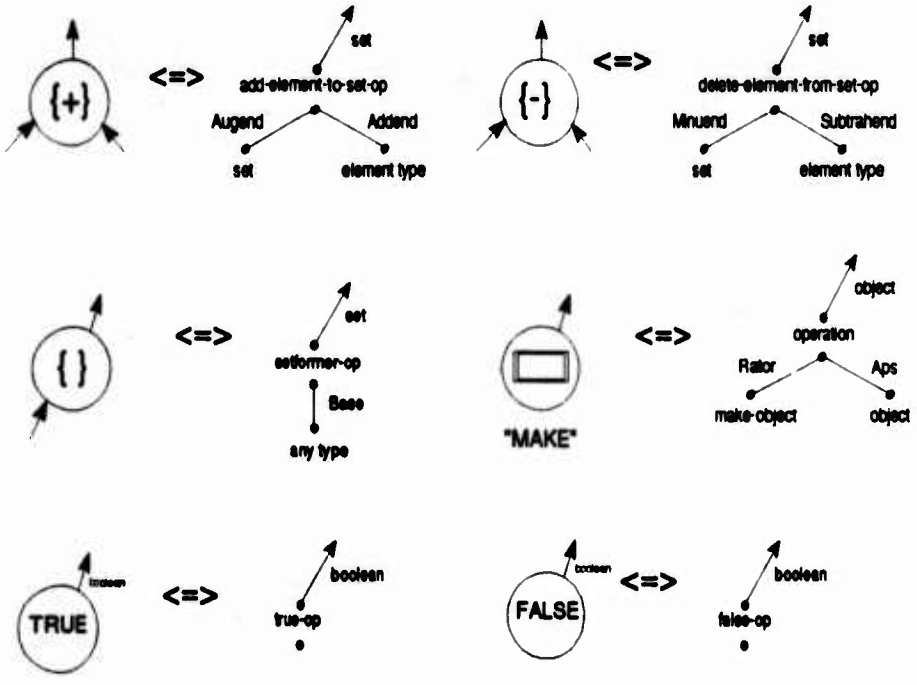


Figure 3.11. Additional Icons Developed for Visual Refine

element subtraction icons, for set operations, has previously been discussed in section 3.3.4 and need not be expanded any further. General set formers are commonly used within Refine formal specifications, and therefore, a way of visualizing this construct needed to be included in the Visual Refine graphical specification language. The object creation, true, and false icons were most likely overlooked during Place's original development of a graphical notation for the Refine wide-spectrum specification language. They are necessary, however, to give Visual Refine a broad enough set of icons for general purpose use.

Finally, definitions for function calls, literal integers and literal strings have been developed. The function icon allows for an easily recognizable means of representing modular formal specifications. The literal definitions are not icons because each integer or string is represented differently. However, facilities for processing literal integers and literal strings has been deemed beneficial to the visualization

process, and thus they are included. Figure 3.12 illustrates the mapping between the Refine AST structure and a visual notation for the above mentioned definitions.

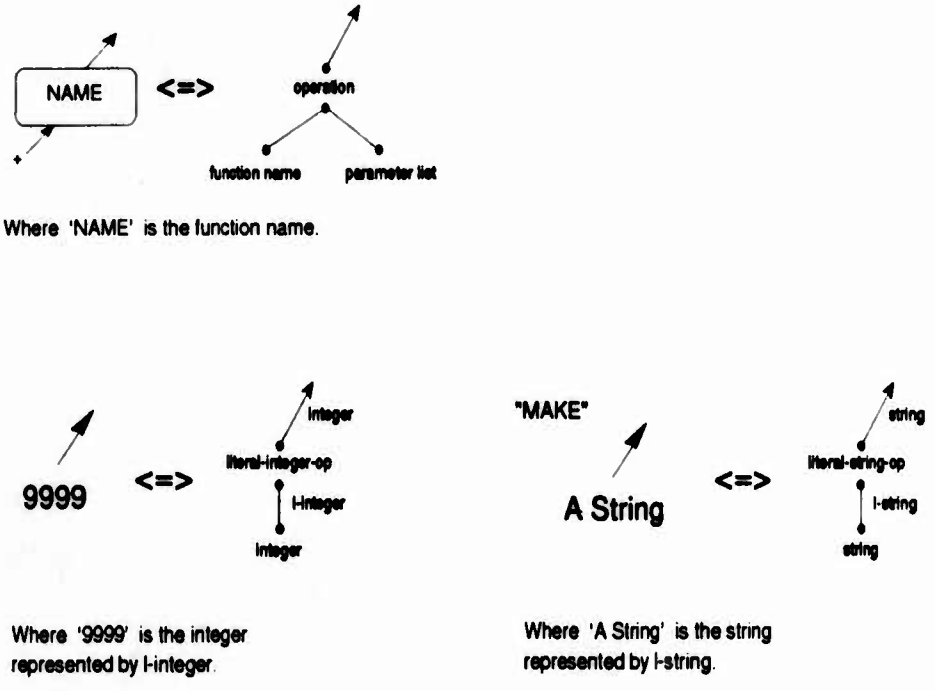


Figure 3.12. Function and Literal Definitions

3.5 Summary

Visual Refine has been defined using a sound methodology. An analysis of the graphical notation, and the intended meaning behind the notation, has been systematically conducted. Each graphical icon, previously developed by Place (15:5-16 - 5-33), was evaluated to determine the most appropriate underlying Refine AST structure. Then each icon was defined to include the input and output ports necessary to represent the underlying Refine AST structure. Next, a formal one-to-one mapping between the formalized version of Place's graphical notation and the underlying Refine AST structure was developed. And finally, several additional icons

were created in order to give the Visual Refine language enough representational capability to be useful.

Appendix E contains a complete and formal definition for each icon specified in the Visual Refine graphical specification language. This appendix can be used as a reference when visualizing Refine formal specifications.

IV. The Application of Visual Refine

4.1 Introduction

This chapter shows how the Visual Refine formal graphical specification language, developed in Chapter III, is realized, or implemented, using the Refine formal specification environment and wide-spectrum language. Once realized, Visual Refine is then demonstrated using Kemmerer's library problem as presented in (1:ix). The realization process is presented first. This is followed by a description of how to use Visual Refine as a general visualization tool for Refine formal specifications. Next section 4.4 describes the library problem, used as a specific example of Visual Refine's capabilities. And finally, section 4.5, demonstrates Visual Refine by applying the implementation, developed in section 4.2, to the library problem discussed in section 4.4.

4.2 Realization of Visual Refine

Visual Refine, as defined in Chapter III and Appendix E, could be used to evaluate Refine source specifications by manually drawing the resulting Visual Refine specification. However, this would be time consuming and error prone. Therefore, a semi-automated transformation system is constructed as a realization of the Visual Refine definition presented in Appendix E. The first, and most obvious, step is to implement a transformation for each of the Refine operations and their associated icons. Visual Refine, by definition, maps or transforms a Refine operator, represented as an object in the Refine object base, to a graphical icon. Because of this relationship, the Refine 'transform' language construct is ideal for constructing Visual Refine transformations. Furthermore, the Refine 'rule' provides a mechanism for naming transforms (19:3-167), and therefore, each Visual Refine mapping is represented as a single Refine rule, consisting of a single Refine transform.

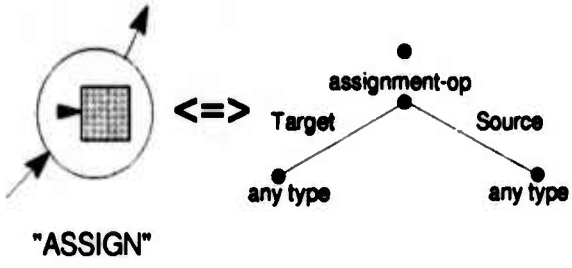
To better understand this relationship several examples from five classes of Visual Refine Icons are provided. The first class, top level icons, specifies the construction of icons which are always constructed at the top level. The next class, embedded icons, specifies the construction of icons which are always constructed as an input to a higher-level icon. Thirdly, the class of boolean icons is specified. Boolean icons can be found at both the top level and embedded level, therefore they require special consideration. The fourth class, binding icons, is a special set of transformations which specify either data or object icons. And finally, the fifth class, miscellaneous icons, specifies any icons not covered in the above four classes. The next five sub-sections discuss in detail each of the respective icon classes.

4.2.1 Top Level Icons Icons in this class represent root nodes within Refine abstract syntax trees. They are referred to as top level icons because they do not require the prior processing of a parent object, or node. Table 4.1 contains the icons belonging to this class. Although the erase-object and display icons are recognized as top level icons, they are discussed and defined as part of the miscellaneous icons, due to a similar underlying AST structure.

Table 4.1. Visual Refine Top Level Icon Class

assignment-op	enumerate-op
erase-object	display-object

Consider the following Visual Refine definition for an assignment statement as an example of icons in this class:



The only condition that must exist prior to the creation of an "ASSIGN" icon, is for an 're::assignment-op' to be located in the Refine object base. The check for this condition is specified as the precondition, or left hand side, of the Refine transform. The creation of the "ASSIGN" icon is the post-condition, or right hand side, of the Refine transform. And finally, this transform is encapsulated in a Refine 'rule', with a single parameter of type 'object' as follows:

```
rule VR-Assignment-Op( X: object )
   re::assignment-op( X )
 -->
        ri::icon( a ) & ri::home-surface( a ) = DS        &
                        ri::icon-type( a ) = 'ri::ellipse &
                        ri::height-width-ratio( a ) = 1.0 &
                        ri::label( a ) = [ "ASSIGN" ]     &
                        ri::clip-icon-label?( a ) = false &
                        VR-Object-for-Icon( a ) = X
```

Functionally, the 'assignment-op' sets the value of the 'target' attribute equal to the value of the 'source' attribute. No result is produced after carrying out this action; therefore, processing this icon does not require a link to be produced between this icon and its parent icon. Thus this icon is always evaluated and produced at the top level. Control links, or dashed lines, will be added later to show how top level icons interrelate. Similarly, the 'enumerate-op' is a side effect producing operator.

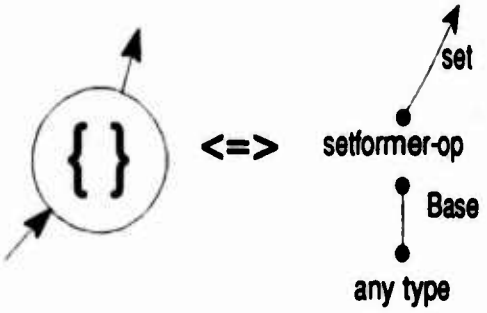
meaning that no result is produced by the operation. The 'enumerate-op' is specified in the same manner as the 'assignment-op'. Appendix F, section F.2, contains complete specifications for icons within this class.

4.2.2 Embedded Icons Embedded icons are icons that are always linked to a higher level parent icon. Prior to creating an embedded icon, the parent icon must first be processed. Table 4.2 contains the icons included in this class.

Table 4.2. Visual Refine Embedded Level Icon Class

setformer-op	literal-integer-op
literal-string-op	plus-op
minus-op	times-op
divided-by-op	div-op
mod-op	integer-to-real-op
union-op	intersection-op
arbitrary-element-op	set-to-seq-op
first-op	prepend-element-to-sequence-op
insert-element-in-sequence-op	reverse-op
last-op	append-element-to-sequence-op
subsequence-op	sequence-concatenate-op
delete-element-from-sequence-op	seq-to-set-op
seq-to-map-op	compose-op
inverse-op	map-to-set-op
closure-under-op	transitive-closure-op
get-field-op	filter-op
size-op	domain-of-definition-op
range-op	image-op
add-element-to-set-op	delete-element-from-set-op
true-op	false-op

As an example of how each of the above icons is specified, consider the following definition for the 'setformer-op' icon:



This icon is used to represent either the literal or the general set former operators in Refine (19:3-58 – 3-64). In both cases, the set former is a parameter to another Refine construct, and therefore always produces an embedded icon within Visual Refine. The necessary preconditions for the transform include:

1. the current object be a 'setformer-op' or a 'literalset-op' and
2. the parent object must have been processed prior to processing the 'setformer-op' object.

To determine if the second precondition is met, the attribute 'VR-Icon-for-Object' must be defined for the parent object. If these conditions are met, then the transform produces a "SET" icon and a solid line link from the newly created icon to the icon which represents the parent object. The transform just described is encapsulated within a Refine rule as shown:

```
rule VR-SetFormer-Op( X: object )
   ( re::setformer-op( X ) or re::literalset-op( X ) ) &
   VR-Icon-for-Object?( re::parent-expr( X ) )
 -->
        ri::icon( a ) & ri::home-surface( a ) = DS        &
                        ri::icon-type( a ) = 'ri::ellipse &
                        ri::height-width-ratio( a ) = 1.0 &
                        ri::label( a ) = [ "SET" ]        &
                        ri::clip-icon-label?( a ) = false &
                        VR-Object-for-Icon( a ) = X       &
```

```
                    VR-Icon-for-Object( X ) = a        &
ri::link( b ) &     ri::home-surface( b ) = DS         &
                    ri::source( b ) = a                &
                    ri::target( b ) =
                       VR-Icon-for-Object(
                          re::parent-expr( X ))        &
                    ri::dynamic?(b) = true
```

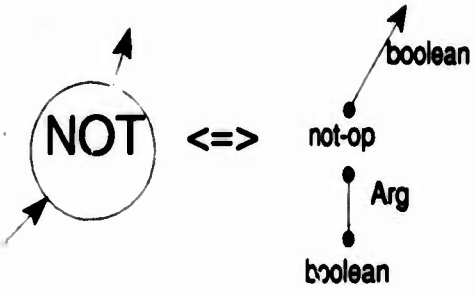
Each icon within this class is specified in the same manner. See Appendix F, section F.3, for a complete specifications of each icon listed in table 4.2.

4.2.3 Boolean Icons The boolean icon class consists of icons which always produce a boolean result. This result can be interpreted as either a top level control construct, or as an embedded level input to some higher level icon. Table 4.3 contains all the icons included in the boolean icon class.

Table 4.3. Visual Refine Boolean Icon Class

lesserorequal-op	lesser-op
greaterorequal-op	greater-op
not-op	implies-op
and-op	ordered-and-op
or-op	ordered-or-op
forall-op	thereexists-op
subset-op	equal-op
member-op	empty-op

The boolean 'Not' operator is used, as an example, to describe icon specifications for this class. The Visual Refine definition for the "NOT" icon is as follows:



Because boolean operators can be found at both the top and embedded levels two separate specifications are required for each icon. In the first case, the icon is at the top level, meaning that the operator is not part of a higher level construct but rather is the highest most, or root, construct being specified. The precondition for the 'Not' operator, in the example, requires only that the current object be a 'not-op'. If this is true, the post-condition consists only of the creation of the "NOT" icon. When encapsulated into a Refine 'rule' the Visual Refine top level "NOT" operator is specified as follows:

```
rule VR-Top-Level-Not-Op( X: object )
   re::not-op( X )
 -->
        ri::icon( a ) & ri::home-surface( a ) = DS        &
                        ri::icon-type( a ) = 'ri::ellipse &
                        ri::height-width-ratio( a ) = 1.0 &
                        ri::label( a ) = [ "NOT" ]        &
                        ri::clip-icon-label?( a ) = false &
                        VR-Object-for-Icon( a ) = X       &
                        VR-Icon-for-Object( X ) = a
```

In the second case, the 'Not' operator is embedded within some other, more encompassing, Refine operator. To process this 'Not' operator, a more stringent precondition is required. The current object must not only be a 'not-op', as before, but the parent object must also have been previously processed, so that the parent object has an icon defined. As with the precondition, the post-condition is also more

complex for the embedded boolean operator. Not only must the "NOT" icon be constructed, but a solid line link is constructed from the "NOT" icon to the icon which represents the parent object. The embedded level specification for the "NOT" icon is as follows:

```
rule VR-Embedded-Not-Op( X: object )
   re::not-op( X ) &
   VR-Icon-for-Object?( re::parent-expr( X ) )
 -->
        ri::icon( a ) & ri::home-surface( a ) = DS          &
                        ri::icon-type( a ) = 'ri::ellipse &
                        ri::height-width-ratio( a ) = 1.0 &
                        ri::label( a ) = [ "NOT" ]          &
                        ri::clip-icon-label?( a ) = false &
                        VR-Object-for-Icon( a ) = X         &
                        VR-Icon-for-Object( X ) = a         &
        ri::link( b ) & ri::home-surface( b ) = DS          &
                        ri::source( b ) = a                 &
                        ri::target( b ) =
                           VR-Icon-for-Object(
                              re::parent-expr( X ))         &
                        ri::dynamic?(b) = true
```

Each icon contained within this class requires both a top level and embedded level specification. Appendix F, section F.4, contains complete specifications for all icons within this class.

4.2.4 Binding Icons This section describes the specification of a special class of icons which represent either data or object references. Six Refine rules, each discussed separately, are required to adequately specify these icons. An internal Visual Refine structure (binding-list) is used to maintain a list of all data or object icons which are constructed. This prevents multiple icons from being built for multiple references to a single data or object entity. When an icon is built by a 'binding icon' rule it is inserted into the 'binding-list' structure for future reference. The Visual Refine binding icon rules are now discussed; however the Refine specifications are

not include in this section. The Refine specification for each individual rule may be found in Appendix F, section F.5.

The first Visual Refine binding icon rule discussed is "VR-Binding". When a data entity is first introduced, a 'binding' object is created in the abstract syntax tree. If a 'binding' object is recognized, the object does not already have an icon specified, and an icon with the same name is not already in the 'binding-list' structure, then the preconditions for this rule have been met. If fired, this rule produces a data icon with a label equal to the 'name' attribute of the 'binding' object. Additionally, the data icon is added to the 'binding-list' structure, and a solid line link is constructed from the parent object's icon to the data icon. In some cases a parent icon does not exist, in which case the specification of the link has no effect.

The "VR-Object" Visual Refine binding icon rule is similar to the "VR-Binding" rule. This rule however, is used to build an icon which represents a reference to an object, rather than a data entity. Object references are recognized as a Refine 'operation' operator, where the 'rator' attribute, of the operation object, refers to an 'Object-Class'. Therefore, the preconditions, for this rule to fire, include the recognition of a Refine 'operation' object and the determination that this 'operation' refers to an 'Object-Class'. If fired the rule produces a Visual Refine object icon where the label is built from the 'bindingname' attribute of the 'rator' attribute of the 'operation' object. This object icon is added to the 'binding-list' structure and a solid link is constructed from the object icon to the icon representing the parent object.

When specifying a reference to an attribute of an object in Refine, an 'operation object is created in the abstract syntax tree. This 'operation' is distinguished from other 'operation' objects by evaluating the 'rator' attribute. If the 'rator' attribute refers to a 'powermapping-op' then an attribute of an object has been referenced. The next two rules discussed, "VR-Data-With-Object-Link" and "VR-Data-Without-Object-Link", transform these types of references into Visual Refine icons and links. Ideally, attribute reference icons should be connected not only to

the icon representing the parent object, but also to the object icon for which the attribute belongs. This additional link is the difference between the above mentioned rules. If the object icon has been previously constructed, then the link to the object icon can be made; otherwise, this link is not shown, due to lack of sufficient information for its construction.

The last two rules in this class, "VR-Binding-Ref-Build-Icon" and "VR-Binding-Ref-Link-Icon", are combined to handle references to previously bound variables. Since it is common to make multiple references to the same variable within a formal specification segment, it is desirable to only produce the data icon once and then just link subsequent references to this icon. The 'binding-list' structure is used to determine whether or not the data icon has been built.

4.2.5 Miscellaneous Icons Four icons remain to be defined. They are grouped together to form a miscellaneous class of icons. Table 4.4 contains the icons included in this class.

Table 4.4. Visual Refine Miscellaneous Icon Class

Nth-element-op	erase-object
make-object	display-object

Each of the above operators is represented in the Refine abstract syntax tree as an 'operation' object. The Nth-Element-Op operator must have had the sequence previously specified and an icon constructed for this sequence, prior to processing a reference to the Nth element of the sequence. The 'binding-list' structure is once again used to make this check. Each of the other tree operators in this class are determined by checking the 'bindingname' attribute of the 'rator' attribute, in the 'operation' object. The 'erase-object' and 'make-object' have the same respective 'bindingname'; the Display-Object-Op is determined by a 'bindingname' of "format", representing the Lisp language 'format' statement, used for output in Refine.

Complete specifications for the above miscellaneous icons are found in Appendix F section F.6.

4.3 Using Visual Refine

An environment for transforming Refine specifications has been developed. Each rule, described above and listed in Appendix F, represents the transformation of a particular Refine object into a specific Visual Refine icon. Additionally, it is presumed that a Refine specification exists for a given application. Figure 4.1 illustrates that both the Visual Refine transformation system and the Refine application are represented in the Refine object base. What appears to be missing, however, is

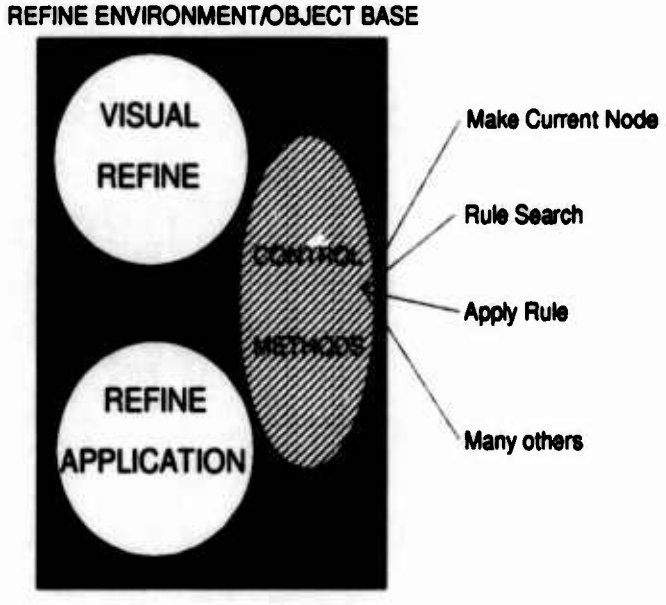


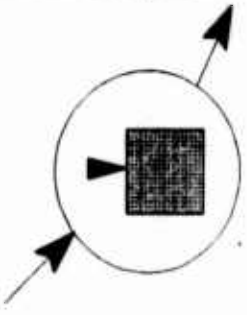
Figure 4.1. Refine Object Base

the controlling mechanisms required to connect, or apply, the Visual Refine system to the Refine application. One method of providing these control mechanisms is to explicitly develop control functions for the Visual Refine transformation system; however, this is not necessary because of the powerful object base manipulation features found in the Refine environment. Three capabilities are of particular interest. They are:

1. selection of a particular object (the MCN function),
2. determination of what rules in the object base could potentially apply to the selected object (the Rule Search function), and
3. the application of a particular rule to the selected object (the Apply Rule command).

Each of these three commands is described further below. Additional object base manipulation commands can be found in (19).

4.3.1 Generation of Textual Icons The motivation for this research is to show the applicability of visual technologies toward making formal specifications easier to understand. This research is not concerned with the implementation issues associated with constructing detailed graphical icons. For this reason, some icons are annotated with a word contained in quotes. This word represents a simpler textual icon, created for the Visual Refine prototype system, developed as part of this research. For example, the Visual Refine assignment icon,



is prototyped as a textual icon that looks like

Textual icons are more easily implemented, as compared to the detailed graphical icons defined for many of the Visual Refine constructs. Therefore, the simpler tex-

tual icons have been used in the Visual Refine prototype system, and are shown in the detailed examples found in section 4.5. The final diagram in each example, and all of the visualizations in Appendices H and J, are depicted using the graphical icons of Visual Refine. To avoid confusion, the textual annotation has been included, as part of the icon definition. Visual Refine icon definitions which do not have the textual annotation have been prototyped as defined.

4.3.2 Visualization Process Visual Refine is a semi-automated visualization tool, used to visualize formal specifications defined using the Refine wide-spectrum specification language. Specifically, Refine functions and rules are considered the base, or root, objects of interest. And commonly used Refine operators are the elements, or leaf objects, used to build up the visual formal specification. In order to visualize a Refine function or rule, the user must first have specified the desired behavior using the Refine surface syntax described in (19:Chapter 3). Then this specification is compiled into the Refine object base, where it can be processed. To begin the visualization process, the Visual Refine system must be loaded as part of the Refine environment. This is accomplished by loading the following files:

1. vrefine.fasl4,
2. vrtop.fasl4,
3. vrembed.fasl4,
4. vrbooln.fasl4,
5. vrbind.fasl4, and
6. vrmisc.fasl4.

Next, a graphical window must be initialized. This is accomplished by typing the command "(visual-refine)" at the Refine command interface. A graphical window will appear with a "STOP" icon defined in the lower right corner of the window (see figure 4.2). Additionally, the desired Refine function or rule must be defined as the

current object. This is accomplished using the 'make current node' (mcn) command (19:4-58). Type "(mcn 'desired-name)" at the Refine command interface, where desired-name is the function or rule to visualize. The Visual Refine system is now ready for use.

Because Visual Refine is implemented using Refine rules, the Refine environment can be used to determine what actions are possible at each step of the visualization. To make this determination, the Refine rule search (rs) command is used (19:4-66). This command searches the Refine object base for all rules which might apply to the current object. Once these applicable rules are determined, the Refine apply rule (ar) command (19:4-66) is used to select the desired action. Each node in the Refine AST is processed using this approach. Several commands, described in the Refine Users Manual (19:4-57 - 4-61), are available for traversing the AST.

Dashed line control arcs are added to the diagram as the final step. The Refine intervista subsystem provides basic tools for manually creating icons and arcs (18:7-15 - 7-17). In the case of the control arcs, the user moves the mouse cursor over the desired source icon and clicks the mouse button. An "Icon Actions" menu appears, from which the "Link to Target" option is chosen. Then place the mouse cursor over the target icon to produce the link. After the link is produced, it must be edited so that it has the desired attributes of a control link. First, place the mouse cursor over the link and click the mouse button; this activates a "Link Actions" menu. Select the "Change Type" option, and then the "PIECEWISE-LINEAR-DASHED" type. Next select the link again. This time choose the "Edit Label" option and delete the string "New Label". Continue this process until all necessary control links are created. Once the control links are in place, the visualization is complete.

A general methodology for using the Visual Refine formal transformation system is as follows:

1. Initialize the Visual Refine environment. This includes opening and initializing a graphics window, and selecting the Refine function or rule to visualize. The desired function or rule is identified as the current node.
2. Perform a rule search. Typically only one rule applies, however, if multiple rules apply, the user must determine which, if any, of the rules apply.
3. Apply the desired rule and determine which node should be processed next. If the current node has children, select an appropriate child. If the current node is a leaf node, return upward in the AST until a node is found whose children have not been completely processed; select an unprocessed child.
4. Manually position any previously created icons, if desired. Then repeat steps two, three, and four until the entire AST has been processed.
5. Manually insert any necessary control links.

4.4 The Library Problem

Kemmerer originally developed the library problem as an example system for his 1985 article, "Testing Formal Specifications to Detect Design Errors" (13:34). In 1987, the Fourth International Workshop on Software Specification and Design included Kemmerer's library problem in the problem set for the workshop, and defines the problem as follows (1:ix):

> Consider a small library database with the following transactions:
>
> 1. Check out a copy of a book / Return a copy of a book;
> 2. Add a copy of a book to / Remove a copy of a book from the library;
> 3. Get the list of books by a particular author or in a particular subject area;
> 4. Find out the list of books currently checked out by a particular borrower
> 5. Find out what borrower last checked out a particular copy of a book.

There are two types of users: staff users and ordinary borrowers. Transactions 1, 2, 4, and 5 are restricted to staff users, except that ordinary borrowers can perform transaction 4 to find out the list of books currently borrowed by themselves. the data base must also satisfy the following constraints:

1. All copies in the library must be available for checkout or be checked out.
2. No copy of the book may be both available and checked out at the same time.
3. A borrower may not have more than a predefined number of books checked out at one time.

From the above problem definition, Place developed a Refine formal specification of Kemmerer's library problem (15:Appendix F). Because Visual Refine does not provide facilities for the set attribute (set-attr) Refine language construct, Place's specification requires slight modifications before applying Visual Refine. The Refine set-attr language construct is a shorthand notation for assigning values to several attributes of a single object. Another way of viewing this is as a block of simple assignment statements. Therefore, no behavioral change occurs when the set-attr constructs are replaced by a block of simple assignment statements. This modified Refine formal specification for Kemmerer's library problem is contained in Appendix G, and is the Refine source used to demonstrate Visual Refine.

Both a Refine 'rule' and a Refine 'function', from the library problem, are used throughout the remainder of this chapter, to illustrate the process of constructing Visual Refine representations. A representative rule from the library problem is the 'Add-Book-To-Library' rule; this rule has the following Refine specification:

```
rule Add-Book-To-Library( author : string,
                          title : string,
                          subject : set( string ) )
    empty( { b | (b: book) book( b ) &
                          title-of-book( b ) = title }
         ) -->
```

```
let( new-book: book = make-object( 'book ) )
   author-of-book ( new-book ) <- author;
   title-of-book  ( new-book ) <- title;
   subject-of-book( new-book ) <- subject;
   on-shelf       ( new-book ) <- true;
   book-out       ( new-book ) <- false
```

Because of its simple, straight forward nature, the 'Print-Book-Set' function was chosen to illustrate in complete detail the process of constructing a Visual Refine representation. The 'Print-Book-Set' function is specified as follows in Refine:

```
function PRINT-BOOK-SET( set-of-books: set(book) ) =
   if( empty( set-of-books ) ) then
      format( true, "This set of books is empty ~%" )
   else
      enumerate b over set-of-books do
         format( true, "AUTHOR: ~S ~30T TITLE: ~S ~%
                                  ~10T SUBJECT: ~S ~%",
                       author-of-book( b ),
                       title-of-book( b ),
                       subject-of-book( b )
               )
```

4.5 Applying Visual Refine to the Library Problem

This section discusses, in detail, the construction of a Visual Refine representation for a Refine specification. The first example used, is the above simple Refine function, 'Print-Book-Set'. Following this detailed example, the visualization of the 'Add-Book-To-Library' Refine rule is discussed. The visualization of the 'Add-Book-To-Library' rule is presented in less detail; the intent being to illustrate that both Refine functions and rules can be visualized. The number associated with a Visual Refine rule is not necessarily the same from one execution to the next. These numbers are dependent upon the order in which Refine rule definitions are compiled and loaded into the Refine environment. This is one reason why a rule search command is necessary.

4.5.1 Print-Book-Set The following discussion assumes that the user is familiar with the Refine environment and specification language. Additionally, it is assumed that Refine has been started and the Intervista subsystem has been loaded. The following files must also be loaded:

1. vrefine.fasl4,
2. vrtop.fasl4,
3. vrembed.fasl4,
4. vrbooln.fasl4,
5. vrbind.fasl4, and
6. vrmisc.fasl4.

In addition to the above files, the Refine object base must also contain the compiled version of any functions and/or rules which the user desires to visualize. For this example the compiled version of Kemmerer's library problem, contained in the file "library.fasl4", has been loaded from the Refine command line. Once all the above files are loaded, the procedure for visualizing a function or rule can begin. The following detailed example will visualize the 'Print-Book-Set' Refine function, as specified in section 4.4.

The visualization process is begun by typing "(visual-refine)" at the Refine command line prompt, as follows:

```
.> (visual-refine)
RE::*UNDEFINED*
```

This command opens a graphics window, shown in figure 4.2, where the results of visualizing the 'Print-Book-Set' function are displayed.

Figure 4.2. Visual Refine Initial Window

The 'Print-Book-Set' function is represented as an object ir. the Refine object base. This object must be identified as the object of interest. The 'mcn' (Make Current Node) command is used as follows to accomplish this task.

```
.> (mcn 'Print-Book-Set)
function PRINT-BOOK-SET (SET-OF-BOOKS: set(BOOK))
  = if empty(SET-OF-BOOKS)
     then FORMAT(true, "This set of books is empty ~%")
     else enumerate B over SET-OF-BOOKS
        do FORMAT
           (true,
            "AUTHOR: ~S ~30T TITLE: ~S ~%
                          ~10T SUBJECT: ~S ~%",
            AUTHOR-OF-BOOK(B), TITLE-OF-BOOK(B), SUBJECT-OF-BOOK(B))
```

Note that the function, as represented by the object base, is displayed as a result of the 'mcn' command. Figure 4.3 is a pictorial representation of the abstract syntax tree (AST) which represents the 'Print-Book-Set' function.

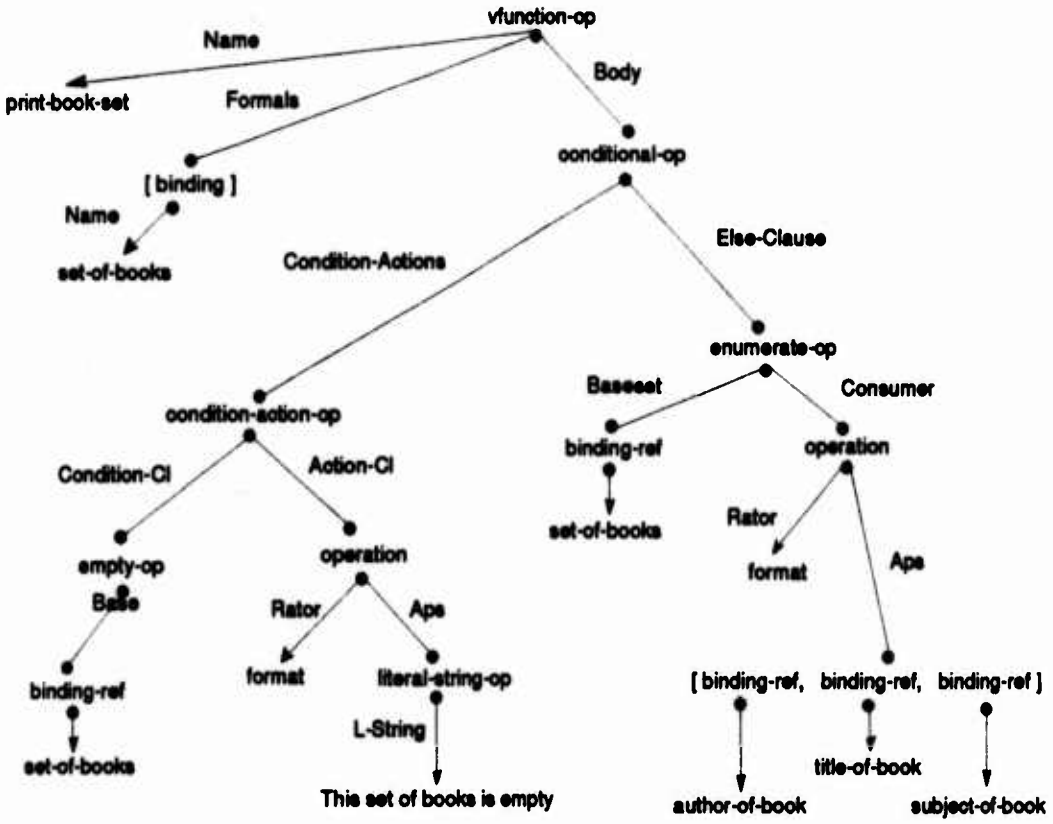


Figure 4.3. Print-Book-Set: Abstract Syntax Tree

At this point, the user of Visual Refine begins finding and applying rules to the nodes, or objects, which form the AST for the 'Print-Book-Set' function. The Refine rule search (rs) command (19:4-66) is used to identify rules which might be applicable to the current object.

```
.> (rs)
- Rules for: function PRINT-BOOK-SET (SET-OF-BOOKS: set(BOOK))
  = if empty(SET-OF-BOOKS)
     then FORMAT(true, "This set of books is empty ~%")
     else enumerate B over SET-OF-BOOKS
        do FORMAT
           (true,
            "AUTHOR: ~S ~30T TITLE: ~S ~%
                          ~10T SUBJECT: ~S ~%",
            AUTHOR-OF-BOOK(B), TITLE-OF-BOOK(B), SUBJECT-OF-BOOK(B)) -
3) VR-VFUNCTION-OP
```

The Refine apply rule (ar) command (19:4-66) is then used to apply the desired rule to the current object.

```
.> (ar 3)
Rule successfully applied.
function PRINT-BOOK-SET (SET-OF-BOOKS: set(BOOK))
  = if empty(SET-OF-BOOKS)
     then FORMAT(true, "This set of books is empty ~%")
     else enumerate B over SET-OF-BOOKS
        do FORMAT
           (true,
            "AUTHOR: ~S ~30T TITLE: ~S ~%
                          ~10T SUBJECT: ~S ~%",
            AUTHOR-OF-BOOK(B), TITLE-OF-BOOK(B), SUBJECT-OF-BOOK(B))
```

The Refine 'pup' command (19:4-30) is used to print the current object in a format where each of the defined attributes for that object can be identified.

```
.> (pup)
PRINT-BOOK-SET - the re::vfunction-op
  re::class:  re::vfunction-op
  name:  print-book-set
```

```
re::vfn-binding: #1<PRINT-BOOK-SET - a re::binding>
re::formals: [#2<SET-OF-BOOKS - a re::binding>]
re::body: #3<a re::conditional-op>
re::lisp-function: (DEFUN PRINT-BOOK-SET (SET-OF-BOOKS) ...)
re::flow-info-for-def: {#4<a re::flow-datum>, #5<a re::flow-datum>, ..}
re::compilations: {#6<a re::compilation>}
re::compiled-okay?: true
re::type-error?: false
re::top-object-read: true
vr-icon-for-object: #7<an ri::icon>
```

By reviewing the attributes the user of Visual Refine is able to determine which attributes must be processed as child nodes in the AST. For a 're::vfunction-op', only the 're::formals' and 're::body' attributes need to be processed (refer to Figure 4.3). For the 'Print-Book-Set' example, the 're::formals' attribute is processed first.

```
.> re::formals
var SET-OF-BOOKS: set(BOOK)
```

By typing the attribute name at the Refine command line, the current object is changed to the object representing the input attribute. The rule search (rs) and apply rule (ar) commands are once again used to identify and select an applicable rule for the new current object, an 're::binding' object.

```
.> (rs)
- Rules for: var SET-OF-BOOKS: set(BOOK) -
81) VR-BINDING
.> (ar 81)
Rule successfully applied.
var SET-OF-BOOKS: set(BOOK)
```

When the attribute being processed is a set or sequence valued attribute, as is the case for 're::formals', the Refine next (nx) and back (bk) commands (19:4-59) are used to traverse each element in the set or sequence. Additionally, if the nth item of a sequence is desired, the integer n can be used to directly traverse to that object.

```
.> (nx)
```

In the above example, the next (nx) command resulted in no object being returned, indicated by the lack of a Refine program segment being printed. Therefore, the entire sequence representing the attribute 're::formals' has been processed. To return to the top level object for the set or sequence, the integer zero is input at the Refine command line.

```
.> 0
function PRINT-BOOK-SET (SET-OF-BOOKS: set(BOOK))
  = if empty(SET-OF-BOOKS)
     then FORMAT(true, "This set of books is empty ~%")
     else enumerate B over SET-OF-BOOKS
        do FORMAT
           (true, "AUTHOR: ~S ~30T TITLE: ~S ~% ~10T SUBJECT: ~S ~%",
            AUTHOR-OF-BOOK(B), TITLE-OF-BOOK(B), SUBJECT-OF-BOOK(B))
```

Figure 4.4 shows the status of the Visual Refine process after the 're::formals' attribute has been processed. All icons, except for the 'STOP' and 'START', are originally placed in the lower left hand corner of the window. The default diagram mouse handler (18:7-15 – 7-16) is used to place the icons wherever desired by the Visual Refine User.

The body of the 'Print-Book-Set' function can now be processed. Traversing to the 're::body' attribute of the "vfunction-op" object (refer to Figure 4.3), and using the rule search (rs) command, to process the new current node, a 'conditional-op' object, results in the following:

```
.> re::body
if empty(SET-OF-BOOKS)
 then FORMAT(true, "This set of books is empty ~%")
 else enumerate B over SET-OF-BOOKS
   do FORMAT
      (true,
       "AUTHOR: ~S ~30T TITLE: ~S ~%
```

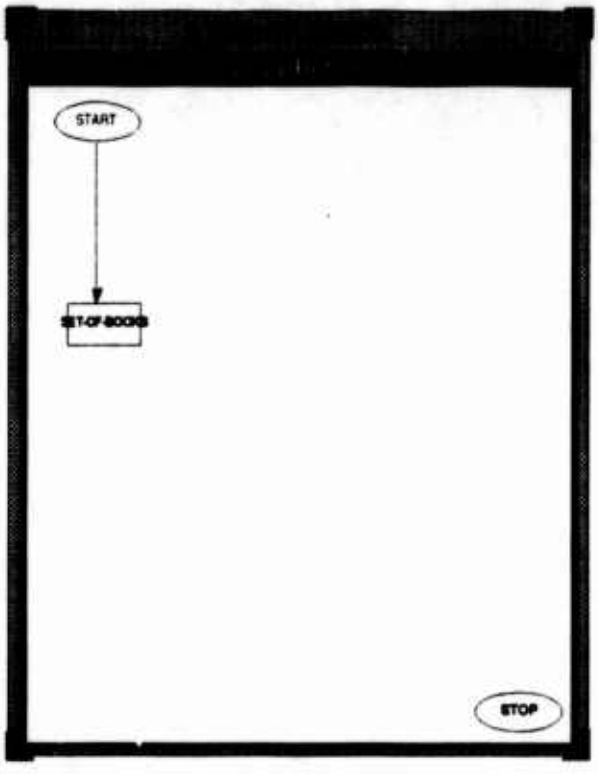


Figure 4.4. Print-Book-Set: After Processing of Formal Parameters

```
                          ~10T SUBJECT: ~S ~%",
        AUTHOR-OF-BOOK(B), TITLE-OF-BOOK(B), SUBJECT-OF-BOOK(B))
.> (rs)
- Rules for: if empty(SET-OF-BOOKS)
 then FORMAT(true, "This set of books is empty ~%")
 else enumerate B over SET-OF-BOOKS
    do FORMAT
       (true,
        "AUTHOR: ~S ~30T TITLE: ~S ~%
                          ~10T SUBJECT: ~S ~%",
        AUTHOR-OF-BOOK(B), TITLE-OF-BOOK(B), SUBJECT-OF-BOOK(B)) -
```

The rule search (rs) command did not provide any applicable rules for this object. Therefore, no Visual Refine processing is required. The (pup) command shows the current object to be a 're::conditional-op', which indeed does not require any processing.

Therefore, the 're::condition-actions' attribute is selected to continue the visualization.

```
.> (pup)
a re::conditional-op
   re::class:  re::conditional-op
   parent-expr: #8<PRINT-BOOK-SET - the re::vfunction-op>
   re::else-clause: #9<an re::enumerate-op>
   re::condition-actions: [#10<a re::condition-action-op>]
```

Once again, applying the rule search (rs) command reveals that the 're::condition-action-op' does not require any Visual Refine processing, so, the (pup) command is again used to select the 're::condition-cl' attribute for further processing.

```
.> re::condition-actions
empty(SET-OF-BOOKS)
then FORMAT(true, "This set of books is empty ~%")
.> (rs)
- Rules for: empty(SET-OF-BOOKS)
then FORMAT(true, "This set of books is empty ~%") -
.> (pup)
a re::condition-action-op
   re::class:  re::condition-action-op
   parent-expr: #3<a re::conditional-op>
   re::action-cl: #11<an re::operation>
   re::condition-cl: #12<an re::empty-op>
.> re::condition-cl
empty(SET-OF-BOOKS)
```

The 'Print-Book-Set' AST has now been traversed to the 're::empty-op' node (refer to Figure 4.3. The rule search (rs) command reveals that processing is required. The 're::empty-op' can be processed as follows:

```
.> (rs)
- Rules for: empty(SET-OF-BOOKS) -
79) VR-TOP-LEVEL-EMPTY-OP
80) VR-EMBEDDED-EMPTY-OP
.> (ar 79)
Rule successfully applied.
empty(SET-OF-BOOKS)
```

The (pup) command reveals that the 're::base' attribute needs to be selected for processing. The selection is made and the attribute is processed in the same manner as before.

```
.> (pup)
an re::empty-op
   re::class:  re::empty-op
   parent-expr: #10<a re::condition-action-op>
   re::base: #13<SET-OF-BOOKS - a re::binding-ref>
   vr-icon-for-object: #14<an ri::icon>
.> re::base
SET-OF-BOOKS
.> (rs)
- Rules for: SET-OF-BOOKS -
86) VR-BINDING-REF-LINK-ICON
.> (ar 86)
Rule successfully applied.
SET-OF-BOOKS
```

The (pup) command is used to show the attributes of the 're::binding-ref' object. No AST attributes remain (refer to Figure 4.3); therefore, processing of the 're::condition-cl' attribute of the 're::condition-action-op' object has concluded. The next step is to return to the top level object and process another attribute. In this case (refer to Figure 4.3), the 're::action-cl' attribute, an 're::operation' object, requires processing next.

```
.> (pup)
SET-OF-BOOKS - a re::binding-ref
   re::class:  re::binding-ref
   parent-expr: #12<an re::empty-op>
   re::bindingname:  set-of-books
   re::ref-to: #2<SET-OF-BOOKS - a re::binding>
.> 0
empty(SET-OF-BOOKS)
.> 0
empty(SET-OF-BOOKS)
then FORMAT(true, "This set of books is empty ~%")
.> re::action-cl
FORMAT(true, "This set of books is empty ~%")
```

The following rule search (rs) reveals several possible applicable rules. The user must decide which rule is most applicable. Since a format statement is used to display text, the 'VR-DISPLAY-OBJECT-OP' rule is chosen.

```
.> (rs)
- Rules for: FORMAT(true, "This set of books is empty ~%") -
82) VR-OBJECT
83) VR-DATA-WITH-OBJECT-LINK
84) VR-DATA-WITHOUT-OBJECT-LINK
87) VR-NTH-ELEMENT-OP
88) VR-ERASE-OBJECT-OP
89) VR-MAKE-OBJECT-OP
90) VR-DISPLAY-OBJECT-OP
.> (ar 90)
Rule successfully applied.
FORMAT(true, "This set of books is empty ~%")
```

To visualize what is displayed, the 're::aps' attribute of the 're::operation' object must be processed further. The text being displayed is the second element in the 're::aps' sequence. It is therefore located and processed.

```
.> re::aps
(true "This set of books is empty ~%")
.> 2
"This set of books is empty ~%"
.> (rs)
- Rules for: "This set of books is empty ~%" -
10) VR-LITERAL-STRING-OP
.> (ar 10)
Rule successfully applied.
"This set of books is empty ~%"
```

No further processing is required for the 're::condition-action-op' attribute of the 're::conditional-op' object (refer to Figure 4.3). Figure 4.5 shows the results of visualization thus far.

The next step is to return to the 're::conditional-op' object and select the 're::else-cl' attribute, an 'enumerate-op' object, for processing (refer to Figure 4.3).

```
.> 0
FORMAT(true, "This set of books is empty ~%")
.> 0
empty(SET-OF-BOOKS)
then FORMAT(true, "This set of books is empty ~%")
.> 0
```

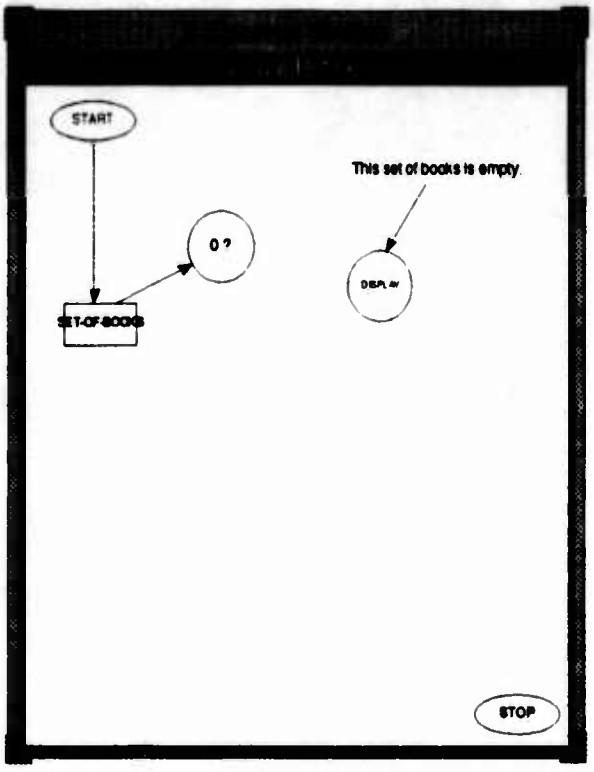


Figure 4.5. Print-Book-Set: After Completion of Conditional Clause

```
if empty(SET-OF-BOOKS)
 then FORMAT(true, "This set of books is empty ~%")
 else enumerate B over SET-OF-BOOKS
    do FORMAT
       (true,
        "AUTHOR: ~S ~30T TITLE: ~S ~%
                              ~10T SUBJECT: ~S ~%",
        AUTHOR-OF-BOOK(B), TITLE-OF-BOOK(B), SUBJECT-OF-BOOK(B))
.> re::else-clause
enumerate B over SET-OF-BOOKS
  do FORMAT
     (true,
      "AUTHOR: ~S ~30T TITLE: ~S ~%
                               ~10T SUBJECT: ~S ~%",
      AUTHOR-OF-BOOK(B), TITLE-OF-BOOK(B), SUBJECT-OF-BOOK(B))
```

As before, the Refine rule search (rs) and apply rule (ar) commands are used to select and apply the appropriate rules to the 're::enumerate-op' object. The traversal and processing of the 're::baseset' attribute of the 're::enumerate-op' object is performed in the same manner that the 're::base' attribute of the 're::empty-op' object was before.

```
.> (rs)
- Rules for: enumerate B over SET-OF-BOOKS
  do FORMAT
     (true,
      "AUTHOR: ~S ~30T TITLE: ~S ~%
                              ~10T SUBJECT: ~S ~%",
      AUTHOR-OF-BOOK(B), TITLE-OF-BOOK(B), SUBJECT-OF-BOOK(B)) -
8) VR-ENUMERATE-OP
.> (ar 8)
Rule successfully applied.
enumerate B over SET-OF-BOOKS
  do FORMAT
     (true,
      "AUTHOR: ~S ~30T TITLE: ~S ~%
                              ~10T SUBJECT: ~S ~%",
      AUTHOR-OF-BOOK(B), TITLE-OF-BOOK(B), SUBJECT-OF-BOOK(B))
.> (pup)
an re::enumerate-op
   re::class:  re::enumerate-op
   parent-expr: #3<a re::conditional-op>
   re::itervar: #20<B - a re::binding>
   re::baseset: #21<SET-OF-BOOKS - a re::binding-ref>
   re::consumer: #22<an re::operation>
   vr-icon-for-object: #23<an ri::icon>
.> re::baseset
SET-OF-BOOKS
.> (rs)
- Rules for: SET-OF-BOOKS -
86) VR-BINDING-REF-LINK-ICON
.> (ar 86)
Rule successfully applied.
SET-OF-BOOKS
.> (pup)
SET-OF-BOOKS - a re::binding-ref
   re::class:  re::binding-ref
   parent-expr: #9<an re::enumerate-op>
   re::bindingname:  set-of-books
   re::ref-to: #2<SET-OF-BOOKS - a re::binding>
```

The (pup) command is used to show the attributes of the 're::binding-ref' object. No AST attributes remain (refer to Figure 4.3); therefore, processing of the 're::baseset' attribute of the 're::enumerate-op' object has concluded. Returning to the 're::enumerate-op' object, the 're::consumer' attribute is chosen to continue processing (refer to Figure 4.3). This is handled in much the same manner as the 'FORMAT' statement above, except the text portion is now represented as three data objects, rather than as a literal text string.

```
.> 0
enumerate B over SET-OF-BOOKS
  do FORMAT
     (true,
      "AUTHOR: ~S ~30T TITLE: ~S ~%
                                ~10T SUBJECT: ~S ~%",
      AUTHOR-OF-BOOK(B), TITLE-OF-BOOK(B), SUBJECT-OF-BOOK(B))
.> re::consumer
FORMAT
  (true,
   "AUTHOR: ~S ~30T TITLE: ~S ~%
                                ~10T SUBJECT: ~S ~%",
   AUTHOR-OF-BOOK(B), TITLE-OF-BOOK(B), SUBJECT-OF-BOOK(B))
.> (rs)
- Rules for: FORMAT
  (true,
   "AUTHOR: ~S ~30T TITLE: ~S ~%
                                ~10T SUBJECT: ~S ~%",
   AUTHOR-OF-BOOK(B), TITLE-OF-BOOK(B), SUBJECT-OF-BOOK(B)) -
82) VR-OBJECT
83) VR-DATA-WITH-OBJECT-LINK
84) VR-DATA-WITHOUT-OBJECT-LINK
87) VR-NTH-ELEMENT-OP
88) VR-ERASE-OBJECT-OP
89) VR-MAKE-OBJECT-OP
90) VR-DISPLAY-OBJECT-OP
.> (ar 90)
Rule successfully applied.
FORMAT
  (true,
   "AUTHOR: ~S ~30T TITLE: ~S ~%
                                ~10T SUBJECT: ~S ~%",
   AUTHOR-OF-BOOK(B), TITLE-OF-BOOK(B), SUBJECT-OF-BOOK(B))
```

Because the three data objects more accurately illustrate the desired output, processing of the third through the last elements in the 're::aps' sequence is more appropriate.

```
.> re::aps
(true
  "AUTHOR: ~S ~30T TITLE: ~S ~%
                                ~10T SUBJECT: ~S ~%"
  AUTHOR-OF-BOOK(B) TITLE-OF-BOOK(B) SUBJECT-OF-BOOK(B))
.> 3
AUTHOR-OF-BOOK(B)
.> (rs)
- Rules for: AUTHOR-OF-BOOK(B) -
```

```
82) VR-OBJECT
83) VR-DATA-WITH-OBJECT-LINK
84) VR-DATA-WITHOUT-OBJECT-LINK
87) VR-WITH-ELEMENT-OP
88) VR-ERASE-OBJECT-OP
89) VR-MAKE-OBJECT-OP
90) VR-DISPLAY-OBJECT-OP
.> (ar 84)
Rule successfully applied.
AUTHOR-OF-BOOK(B)
.> (nx)
TITLE-OF-BOOK(B)
.> (ar 84)
Rule successfully applied.
TITLE-OF-BOOK(B)
.> (nx)
SUBJECT-OF-BOOK(B)
.> (ar 84)
Rule successfully applied.
SUBJECT-OF-BOOK(B)
.> (nx)
```

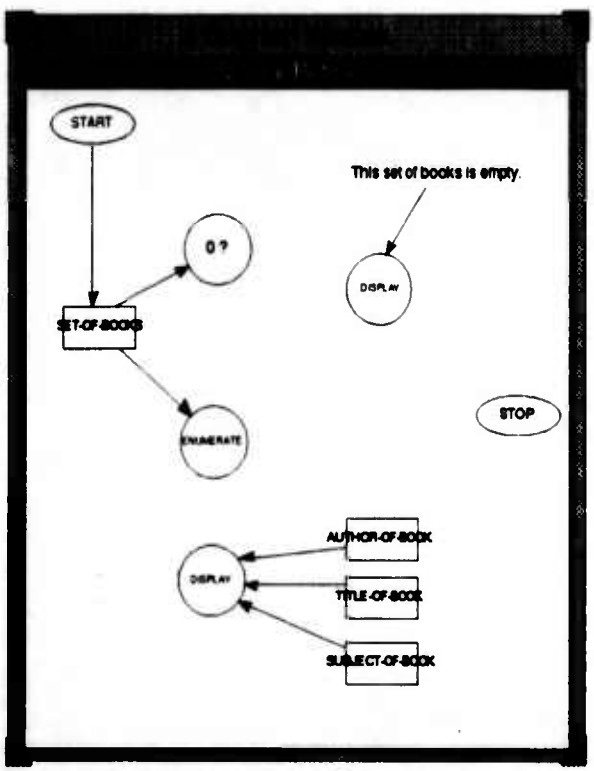


Figure 4.6. Print-Book-Set: Complete Processing of Abstract Syntax Tree

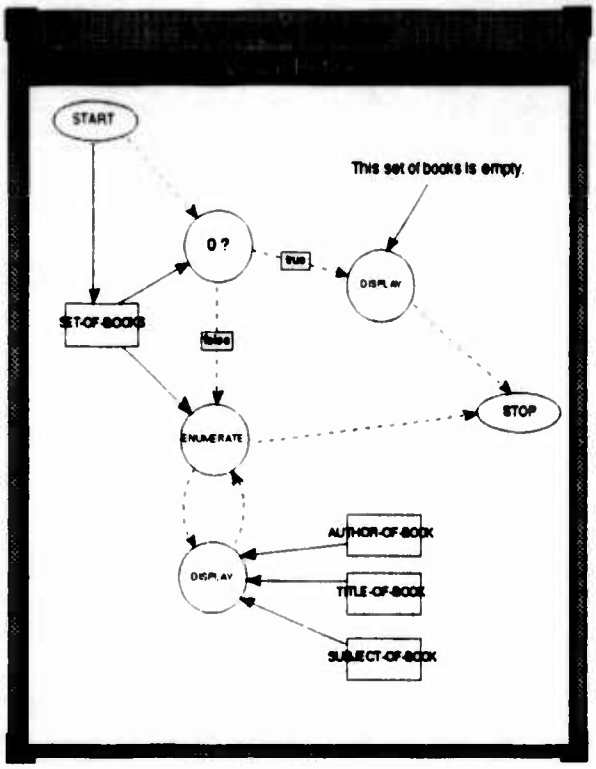


Figure 4.7. Print-Book-Set: Using Visual Refine Implementation Icons

No further processing of the 'Print-Book-Set' AST is required (refer to Figure 4.3). Figure 4.6 shows the state of the Visual Refine processing to this point. The only thing left to make the visualization complete, is the addition of logical control paths. These control paths are provided manually, using the default diagram mouse handler provided by Intervista (18:7-15 – 7-16). A more automated process of constructing the control links could be developed by including processing capabilities for non-icon producing Refine AST operators. This capability has not been included in Visual Refine at this time.

The completed diagram, as represented using the textual implementation icons, is illustrated in Figure 4.7. Figure 4.8 shows what the visualization would look like using the more complicated Visual Refine icons defined in Appendix E.

4.5.2 Add-Book-To-Library The 'Add-Book-to-Library' Refine rule is visualized next, in significantly less detail. This discussion is primarily for the purpose of

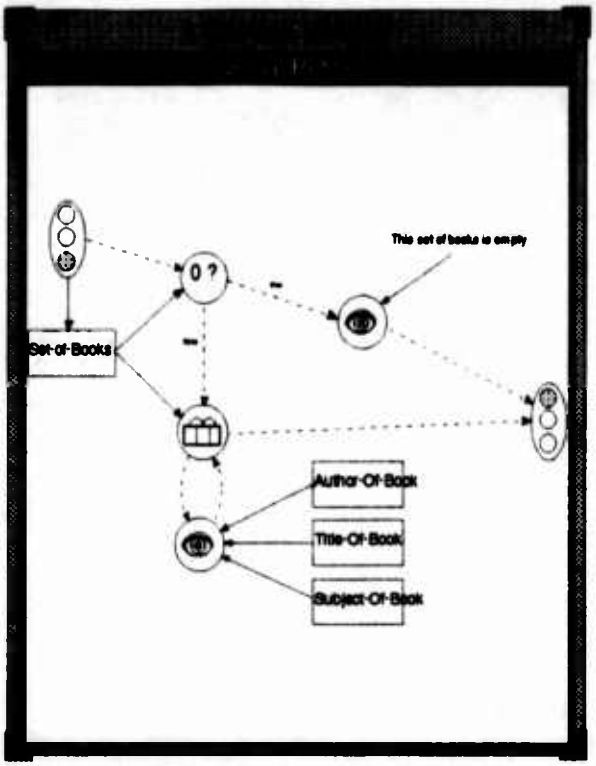


Figure 4.8. Print-Book-Set: Using Actual Visual Refine Icons

demonstrating Visual Refine's capability to visualize rules as well as functions. For that reason, only the beginning steps and final results for visualizing the 'Add-Book-To-Library' rule are presented. To begin, Refine is loaded, including Intervista and Visual Refine, as before. The 'Add-Book-To-Library' rule has also been loaded as part of the library.fasl4 file. As in the case of functions, the Refine make current node (mcn) command is used to define the object of interest as the root node to the 'Add-Book-To-Library' abstract syntax tree.

```
.> (mcn 'Add-Book-To-Library)
rule ADD-BOOK-TO-LIBRARY
  (AUTHOR: STRING, TITLE: STRING, SUBJECT: set(STRING))
  empty
    ({B | (B: BOOK)
        BOOK(B) & TITLE-OF-BOOK(B) = TITLE})
  --> (let (NEW-BOOK: BOOK = MAKE-OBJECT('BOOK))
         AUTHOR-OF-BOOK(NEW-BOOK) <- AUTHOR;
         TITLE-OF-BOOK(NEW-BOOK) <- TITLE;
         SUBJECT-OF-BOOK(NEW-BOOK) <- SUBJECT;
```

```
        ON-SHELF(NEW-BOOK) <- true;
        BOOK-OUT(NEW-BOOK) <- false)
.> (visual-refine)
RE::*UNDEFINED*
```

And finally, like before, the Visual Refine environment is started by typing "(visual-refine)" at the Refine command line. The first difference between functions and rules is noticed during the first issuance of the rule search (rs) command. Instead of the "VR-VFUNCTION-OP" rule being identified, the "VR-RULE-OP" rule is identified. The application of "VR-RULE-OP", however, has the same results as that of "VR-VFUNCTION-OP".

```
.> (rs)
- Rules for: rule ADD-BOOK-TO-LIBRARY
  (AUTHOR: STRING, TITLE: STRING, SUBJECT: set(STRING))
  empty
    ({B | (B: BOOK)
        BOOK(B) & TITLE-OF-BOOK(B) = TITLE})
  --> (let (NEW-BOOK: BOOK = MAKE-OBJECT('BOOK))
        AUTHOR-OF-BOOK(NEW-BOOK) <- AUTHOR;
        TITLE-OF-BOOK(NEW-BOOK) <- TITLE;
        SUBJECT-OF-BOOK(NEW-BOOK) <- SUBJECT;
        ON-SHELF(NEW-BOOK) <- true;
        BOOK-OUT(NEW-BOOK) <- false) -
2) VR-RULE-OP
```

Next a (pup) command is issued to determine what attributes need to be processed. As in the 'vfunction-op', an 're::formals' attribute must be processed, in order to visualize the formal parameter list for the Refine rule. Instead of an 're::body' attribute, the Refine rule has an 're::ruleexpr' attribute which represents the single transform body of the Refine rule construct. The rest of the attributes are not important for visualization.

```
.> (pup)
ADD-BOOK-TO-LIBRARY - the re::rule-op
 re::class:  re::rule-op
 name:  add-book-to-library
```

```
 re::vfn-binding: #1<ADD-BOOK-TO-LIBRARY - a re::binding>
 re::formals: [#2<AUTHOR - a re::binding>, #3<TITLE - a re::binding>, ..]
 re::ruleexpr: #4<a re::rule-impl-op>
 re::lisp-function: (DEFUN ADD-BOOK-TO-LIBRARY (AUTHOR TITLE SUBJECT) ...)
 re::flow-info-for-def: {#5<a re::flow-datum>, #6<a re::flow-datum>, ..}
 re::after-load-forms: {(RE::SET-RULE-APPLICABILITY-INFO 'ADD-BOOK-TO-LIBRARY
'RE::ANY 'RE:*UNDEFINED* '(LAMBDA (AUTHOR TITLE SUBJECT) (NULL (LET
((REVAR::--SETVAR-0 NIL)) (REFINE-LOOP::LOOP REFINE-LOOP::FOR B REFINE-LOOP::IN
(RE::ELEMENTS* (RE::GSET BOOK)) REFINE-LOOP::DO (RE::IF! (RE::STRING-EQUAL-1
TITLE (RE::DB-GET? B 'TITLE-OF-BOOK)) RE::THEN (SETQ REVAR::--SETVAR-0 (RE::IF!
(RE::MEMQ B REVAR::--SETVAR-0) RE::THEN REVAR::--SETVAR-0 RE::ELSE (CONS B
REVAR::--SETVAR-0))))) REVAR::--SETVAR-0))))}
 re::compilations: {#7<a re::compilation>}
 re::compiled-okay?: true
 re::type-error?: false
 re::top-object-read: true
 vr-icon-for-object: #8<an ri::icon>
```

The 're::formals' attribute is processed in exactly the same manner as described for the 'Print-Book-Set' function above. For this reason, the discussion will now turn to the processing of the 're::ruleexpr' attribute. The traversal to this attributes node is made, and a rule search performed to process the 're::ruleexpr' attribute.

```
.> re::ruleexpr
empty
  ({B | (B: BOOK)
      BOOK(B) & TITLE-OF-BOOK(B) = TITLE})
--> (let (NEW-BOOK: BOOK = MAKE-OBJECT('BOOK))
       AUTHOR-OF-BOOK(NEW-BOOK) <- AUTHOR;
       TITLE-OF-BOOK(NEW-BOOK) <- TITLE;
       SUBJECT-OF-BOOK(NEW-BOOK) <- SUBJECT;
       ON-SHELF(NEW-BOOK) <- true;
       BOOK-OUT(NEW-BOOK) <- false)
.> (rs)
- Rules for: empty
  ({B | (B: BOOK)
      BOOK(B) & TITLE-OF-BOOK(B) = TITLE})
--> (let (NEW-BOOK: BOOK = MAKE-OBJECT('BOOK))
       AUTHOR-OF-BOOK(NEW-BOOK) <- AUTHOR;
       TITLE-OF-BOOK(NEW-BOOK) <- TITLE;
       SUBJECT-OF-BOOK(NEW-BOOK) <- SUBJECT;
       ON-SHELF(NEW-BOOK) <- true;
       BOOK-OUT(NEW-BOOK) <- false) -
```

The rule search (rs) command does not identify any rules as being applicable to process this node. Therefore, the (pup) command is used to determine further processing.

```
.> (pup)
a re::rule-impl-op
   re::class:  re::rule-impl-op
   parent-expr: #9<ADD-BOOK-TO-LIBRARY - the re::rule-op>
   re::consequent: #10<a re::bind-op>
   re::antecedent: #11<an re::empty-op>
```

The (pup) command reveals two attributes require processing. The 're::antecedent' attribute, referring to the left hand side of the transform, is processed first.

```
.> re::antecedent
empty
  ({B | (B: BOOK)
      BOOK(B) & TITLE-OF-BOOK(B) = TITLE})
.> (rs)
- Rules for: empty
  ({B | (B: BOOK)
      BOOK(B) & TITLE-OF-BOOK(B) = TITLE}) -
79) VR-TOP-LEVEL-EMPTY-OP
80) VR-EMBEDDED-EMPTY-OP
```

Since no icon was produced for the parent object, a top level icon is needed to begin the processing of the 're::antecedent' attribute. The rest of the nodes in the 're::antecedent' attribute abstract syntax sub-tree are processed in much the same manner as discussed for the 'Print-Book-Set' function. Figure 4.9 shows the results after processing the entire 're::antecedent' attribute.

After the 're::antecedent' attribute is processed, the 're::consequent' attribute needs processing. The traversal is made and a rule search (rs) command issued.

```
.> re::consequent
let (NEW-BOOK: BOOK = MAKE-OBJECT('BOOK))
  AUTHOR-OF-BOOK(NEW-BOOK) <- AUTHOR;
```

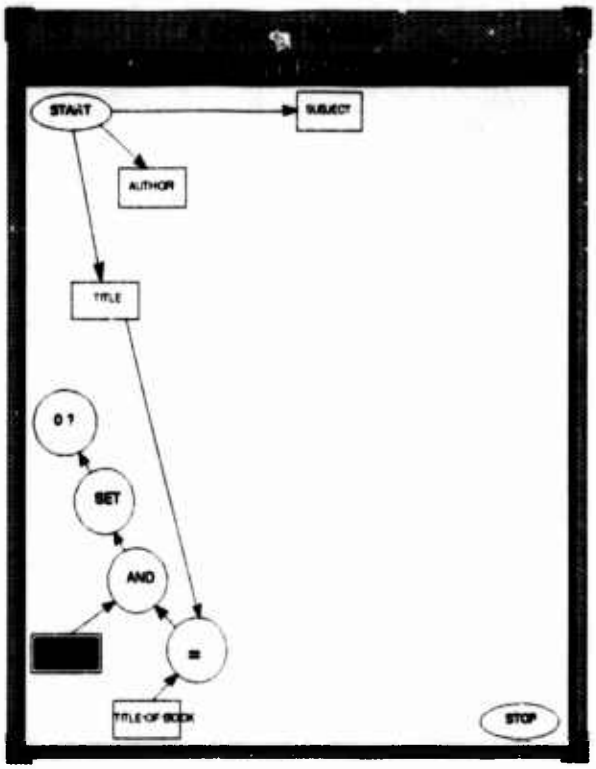


Figure 4.9. Add-Book-To-Library: After Processing of the Antecedent

```
  TITLE-OF-BOOK(NEW-BOOK) <- TITLE;
  SUBJECT-OF-BOOK(NEW-BOOK) <- SUBJECT;
  ON-SHELF(NEW-BOOK) <- true;
  BOOK-OUT(NEW-BOOK) <- false
.> (rs)
- Rules for: let (NEW-BOOK: BOOK = MAKE-OBJECT('BOOK))
  AUTHOR-OF-BOOK(NEW-BOOK) <- AUTHOR;
  TITLE-OF-BOOK(NEW-BOOK) <- TITLE;
  SUBJECT-OF-BOOK(NEW-BOOK) <- SUBJECT;
  ON-SHELF(NEW-BOOK) <- true;
  BOOK-OUT(NEW-BOOK) <- false -
```

Once again the rule search (rs) command fails to provide an applicable rule to process the node. Therefore, the (pup) command is used to determine further processing.

```
.> (pup)
a re::bind-op
   re::class:  re::bind-op
   parent-expr: #4<a re::rule-impl-op>
   re::bindings: {#27<NEW-BOOK - a re::binding>}
   re::body: #28<a re::block-op>
```

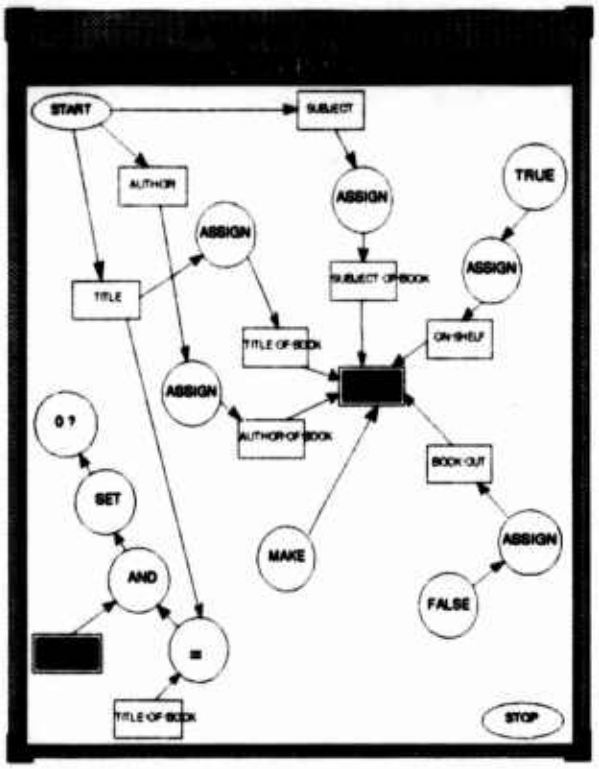


Figure 4.10. Add-Book-To-Library: After Processing of the Consequent

In this case, the 're::bindings' attribute is processed in much the same manner as the 're::formals' attribute discussed above. Following the processing of the 're::bindings' attribute the 're::body' attribute is traversed to and processed.

```
.> re::body
AUTHOR-OF-BOOK(NEW-BOOK) <- AUTHOR;
TITLE-OF-BOOK(NEW-BOOK) <- TITLE;
SUBJECT-OF-BOOK(NEW-BOOK) <- SUBJECT;
ON-SHELF(NEW-BOOK) <- true;
BOOK-OUT(NEW-BOOK) <- false
.> (rs)
- Rules for: AUTHOR-OF-BOOK(NEW-BOOK) <- AUTHOR;
TITLE-OF-BOOK(NEW-BOOK) <- TITLE;
SUBJECT-OF-BOOK(NEW-BOOK) <- SUBJECT;
ON-SHELF(NEW-BOOK) <- true;
BOOK-OUT(NEW-BOOK) <- false -
```

Again, the rule search comes up empty, indicating that further traversal is required before further visualization can occur. The (pup) command shows this to be true.

```
.> (pup)
a re::block-op
   re::class:  re::block-op
   parent-expr: #10<a re::bind-op>
   re::steps: [#29<an re::assignment-op>, #30<an re::assignment-op>, ..]
```

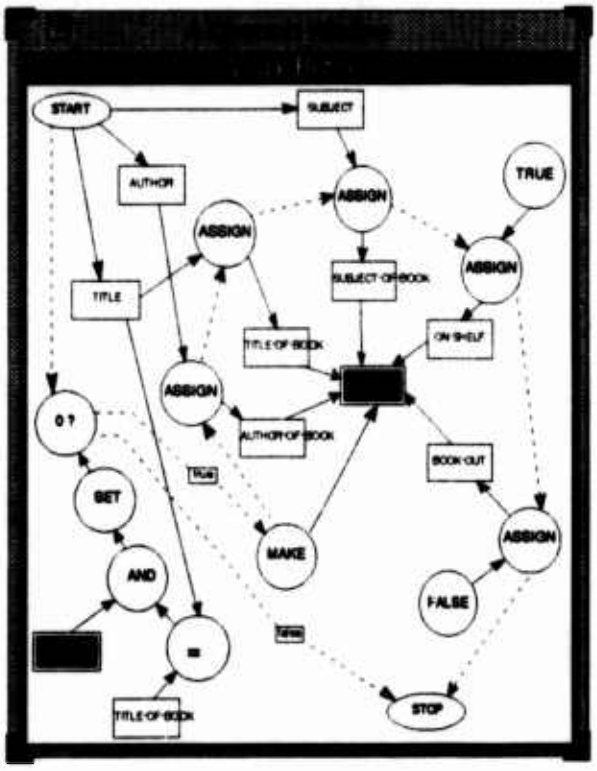


Figure 4.11. Add-Book-To-Library: Intervista Implementation Icons

Processing of the 're::steps' attribute consists of processing a sequence of 're::assignment-op' objects or nodes. This is accomplish in a manner similar to that discussed above, and is therefore not discussed any further. After the 're::steps' attribute has been processed, the 're::consequent' attribute is complete. Figure 4.10 shows the visualization at this point.

The abstract syntax tree for the 'Add-Book-To-Library' Refine rule has now been completely traversed and visualized. The only task remaining to complete the visualization is to add the control links between the top level icons. Figure 4.11 illustrates the completed visualization using the textual icons implemented in Intervista. The Visual Refine visualization is illustrated in figure 4.12. This is what

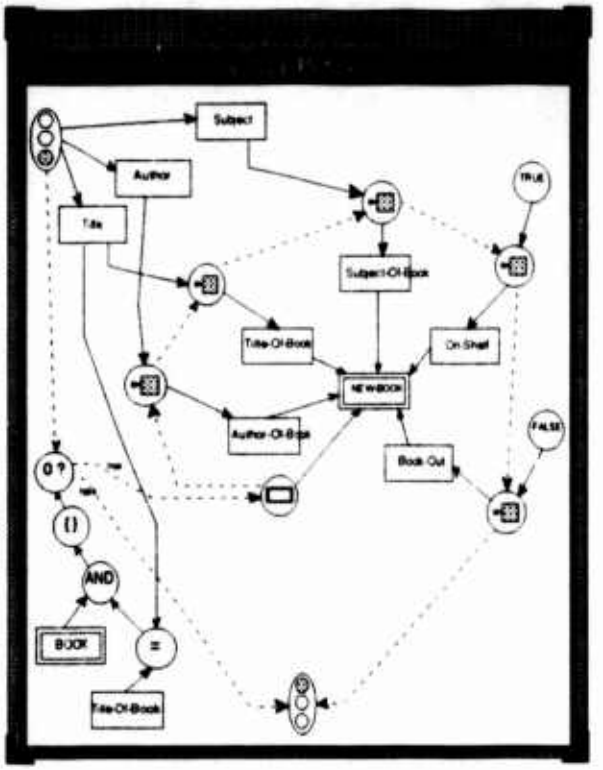


Figure 4.12. Add-Book-To-Library: Visual Refine Visualization

the visualization would look like if the more complicated icons of Visual Refine had been implemented.

4.6 An Automated Approach to Diagram Generation

The above described technique for generating a graph-based visualization of a formal specification requires a significant amount of manual intervention. A knowledge of the underlying abstract syntax tree structure is also required to effectively produce a visualization. This section suggests a methodology which would produce the visualization in an automated manner. Time constraints, prevent the actual implementation of the described methodology.

The Refine 'preorder-transform' function (19:3-195, 3-196) provides a natural means of automating the visualization process. Given an object, representing the root node of an abstract syntax tree, and a sequence of rules, representing the

transformation system, the 'preorder-transform' function will apply the rules to the abstract syntax tree in a top-down fashion. If a rule is successfully applied, the 'preorder-transform' function begins again, starting with the object that was produced by the previously fired rule. (19:3-195) In order to automate Visual Refine, using this function, two tasks must be accomplished:

1. Each rule must be modified to include a check on the left hand side of the transform, which determines if the current object has already been processed. This check is required to prevent the 'preorder-transform' function from firing the same rule over and over without traversing to a child node in the abstract syntax tree.
2. An analysis of the Visual Refine transformation system must be made to determine if two or more rules can be applicable at any one time. If it is not possible to define the rules so that only one rule is applicable at any given time, then a determination must be made as to which rule should come first in the sequence of rules.

By accomplishing the above tasks, an automated diagram generation is possible. However, the layout of the diagram would still require a manual process. To completely automate the visualization process, an automatic diagram layout algorithm needs to be developed as well. The algorithm should take as inputs, a set of icons and a set of links, and produce as output the same sets with modified position attributes. A rudimentary diagram layout algorithm was attempted by this research without success. Even an unsophisticated layout rapidly became too complicated a task to be accomplished within the scope of this research. Therefore, the development of a diagram layout algorithm is left as an interesting future research item.

4.7 Summary

The Visual Refine language, as defined in Appendix E, has now been implemented. Each icon defined in Appendix E has been implemented as one or more Refine transformations. Together, this set of transformations form the Visual Refine graphical formal specification language, or transformation system. Given a Refine formal specification, such as Place's solution to Kemmerer's library problem, Visual Refine transforms the complex mathematical structures found in the Refine language into a more easily understood visual representation. Concurrent research by Blankenship (6:Appendix G) has provided an object oriented decomposition and solution to Kemmerer's library problem. Appendices I and J provide an alternative Refine solution and Visual Refine representation respectively. This alternative is included as another example of Visual Refine's capabilities.

V. Domain Specific Languages

5.1 Introduction

Chapters III and IV have provided a methodology for visualizing formal languages, using the Refine specification environment. Although the visualization of Refine is useful, it is accomplished at a low level of abstraction; representing Refine language constructs, rather then "real world" objects and relationships. To be of greatest use, this newly developed visualization technology, needs to be applied at higher levels of abstraction. One way of achieving more appropriate levels of abstraction, is through the use of Domain Specific languages. Domain Specific languages describe systems using terminology from the domain of interest; thus, visualization of constructs from within this language results in icons which represent "real world" items of interest. Therefore, Domain Specific languages naturally provide an appropriate level of abstraction for describing systems from within a certain domain.

To make this new visualization technology more beneficial, an object-oriented Domain Specific language needs to be developed. This language, when parsed, must result in an abstract syntax tree representation in a formal object base such as the one provided by Refine. Once the language is represented in the object base, the previously developed visualization technologies can be used to develop and implement a visual language representation, and visual transformation system, for the Domain Specific language; thus, a visualization at an appropriate level of abstraction. Section 5.2 discusses the application of this concept to Domain Specific Software Architectures. This is followed by a discussion in section 5.3 on a general methodology for visual support for a Domain Specific Software Architecture.

5.2 Domain Specific Software Architecture

Domain Specific Software Architectures (DSSA) have great potential as a means of establishing a framework for performing object-oriented requirements anal-

ysis, specification, and design. Typically, DSSA's do not provide a formal means of capturing the requirements analysis, specification, and design information. Also DSSA's do not provide a means of manipulating the information in a formal way. To address these shortcomings, the DSSA should be modeled within a formal environment, such as Refine. A formal Domain Model, along with a formal Object Specification language, is used to capture both the structure and behavior of DSSA software components. The formal environment is then used to manipulate the captured information in various ways. By imposing formalism upon the DSSA, this promising methodology can be greatly enhanced in a powerful way (4).

To begin construction of this formal DSSA, a general configuration must be determined. Figure 5.1 illustrates a general configuration developed in support of this and future research. To capture the structure and behavior of software components, the software engineer describes (specifies) each component using the Object Specification language. The Object Specification language is part of the Domain Model, which transforms the input specification into an object base representation. The Domain Model also is used to capture the interrelationships between various aspects of the DSSA; thus the DSSA Formalized Object Base has effectively captured the structure and behavior of the DSSA including the software (domain) components, relationships between components, and relationship within the DSSA.

Through the use of an object base manipulation language, information contained in the DSSA Formalized Object Base can be transformed in three important ways. First, the software component specifications can be transformed into Ada programs, via correctness (behavior) preserving formal transformations. The Develop Ada Components box in figure 5.1 represents this transformation system, with the Ada Component Library being the results of such a system. Another important transformation that can occur is the composition of various Ada components into an Ada application system. By specifying desired behavioral properties of the application system, the structural and behavioral information in the object base can

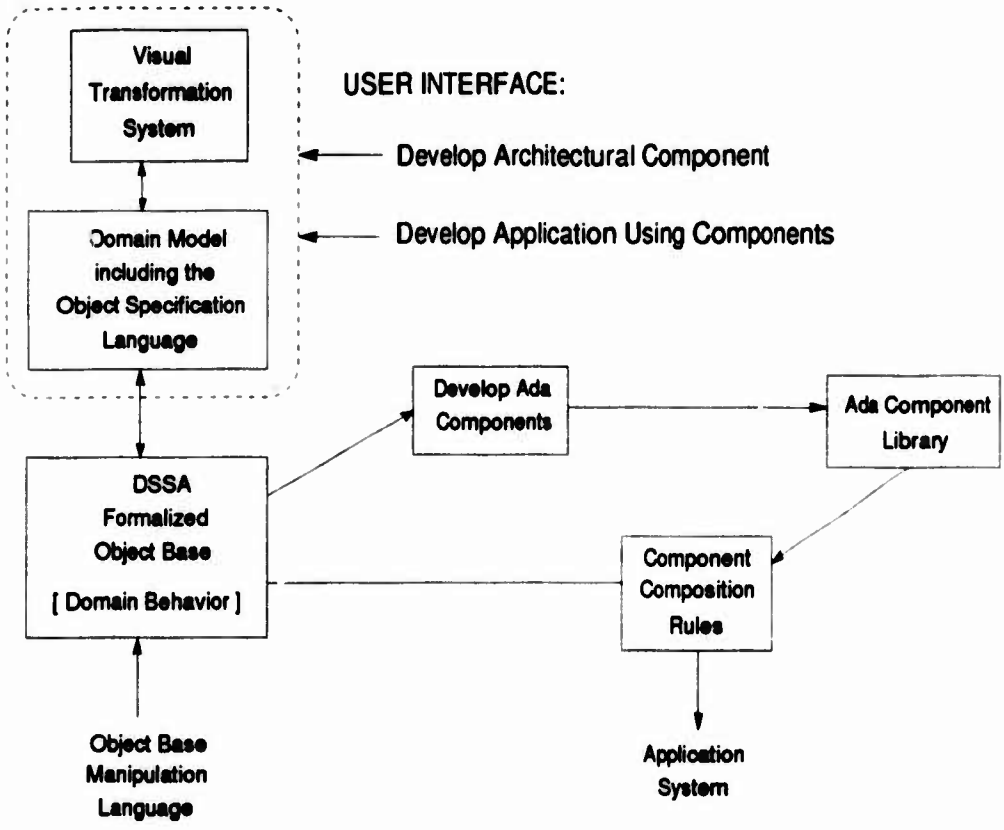


Figure 5.1. General Configuration for Formalizing a DSSA

be used to determine the Ada components necessary to construct the desired application system. These components can then be placed together via a driver program to form the application system. This second transformation system is represented by the Component Composition Rules box in figure 5.1. The final transformation system of interest is the visualization of the DSSA Formalized Object Base. This transformation constructs graph-based diagrams from the object base information using the techniques developed in Chapters III and IV.

5.3 Domain Modeling with Visualization

The Domain Model and Object Specification Language are essential elements in the process of formalizing a DSSA. Time constraints prevent current research efforts from completely developing the Domain Model; however, a methodology for

developing this Domain Model is provided. Additionally, the visualization technology developed previously is shown to be applicable towards enhancing the Object Specification Language required for the proposed Domain Model. This section discusses the necessary steps towards building a useful Domain Specific Modeling System. The following three main tasks must be accomplished:

1. Construct a Domain Model and Language.
2. Implement the Domain Model.
3. Develop the Visual Transformation System.

5.3.1 Domain Model Construction Before a Domain Specific Modeling System can be constructed, a Domain Model and Object Specification Language must be developed. Two approaches are offered:

1. Develop the Domain Model and Object Specification Language, from the ground up, specifically for the needs of the desired Domain Specific Modeling System.
2. Develop the Domain Model from an existing Object Specification Language, adapting the existing language as necessary to meet the requirements of the desired Domain Specific Modeling System.

Both of the above approaches have merit, and each has its shortcomings. A short discussion of each approach follows.

Development of a Domain Model and Language specifically for the needs of the desired Domain Specific Modeling System has the advantage of being constructed to meet specific features and requirements of the system. For example, software objects could be modeled with specific attributes to reflect structural and behavioral properties of the object. The development of the Domain Model for such a system could most likely be accomplished without too much difficuitly; however, the development of a grammar for the Object Specification Language, which supports the developed

Domain Model, is not an easy task. An Object Specification Language is essential to the Domain Specific Software Architecture model presented above; making the choice of this approach extremely risky.

An alternative to building the Domain Model and Language from the ground up, is to construct a Domain Model based upon an existing Object Description Language. The main advantage to this approach is the significant reduction in development time and risk. The primary disadvantage is that desired features of the Domain Specific Modeling System may need to be defined in an awkward manner or possibly even eliminated. As long as an existing language can be found which adequately describes both the software components within the Domain Specific Software Architecture and the interrelationship between components then the reduced development time and risk makes this the recommended option. A language which potentially meets the above requirements is the Requirements Modeling Language (RML) developed and specified in (12).

Given that an existing Object Definition Language, like RML, has been determined to meet requirements of the Domain Specific Modeling System, then four things need to be defined (17:4-1):

1. Define what the abstract syntax trees will look like.
2. Define object classes for each type of node in the abstract syntax trees.
3. Define attributes that capture the structure of the abstract syntax trees.
4. Define attributes for any annotations that will be needed.

Once these items are defined, the Domain Model is ready to be implemented as discussed below.

5.3.2 Domain Model Implementation The Dialect^{TM} subsystem of the Refine specification environment provides an excellent means for implementing the Domain Model (17:4-1 – 5-28). The nodes of the abstract syntax tree definitions developed

for the Domain Model are represented in Refine as object classes; whereas the tree structure and annotations are represented as object attributes. Each object class defined in step 2 above, is converted to a Refine definition by declaring the class as a subtype of either the Refine predefined super class (user-object) or as a subtype of a previously defined class (17:4-3, 4-4). Next the attributes are declared. Each attribute represents an arc in the abstract syntax tree. These attributes are used in production definitions, discussed later. To complete the implementation of the Domain Model, the abstract syntax tree attributes must be distinguished from the annotation attributes. This is done through the use of the Refine "define-tree-attributes" function (17:4-5).

The Domain model describes the semantics of the Object Specification Language and forms a basis from which a language grammar can be defined. The Dialect subsystem is also used for describing grammar productions, which are written using an extended Backus Naur Form (BNF), described in (17:5-1 – 5-28). By using attribute names, defined in the Domain Model, as building blocks within the grammar productions, Dialect is able to automatically construct abstract syntax tree representations of the Object Specification Language in the Refine object base. This feature of Dialect allows the user to specify only the surface syntax, and Dialect generates most, if not all, of the necessary semantic routines, typically associated with a compiler.

5.3.3 Visualizing the Domain Model The Object Specification Language implemented using the above technique can now be used to describe the software components, including structure and behavior. As these descriptions are compiled, they will be represented as abstract syntax trees in the Refine object base. The visualization development methodology described in Chapters III and IV can then be directly applied. First, define which structures are of interest for visualization. Refine's pattern matching facilities make it possible to recognize a single node in the abstract syntax tree, or to recognize a particular abstract syntax structure, consisting

of several nodes. This ability to group nodes together make it possible to visualize a wide range of structures, from the very detailed to the more abstract. Next, develop an icon for each structure of interest, and write Refine rules to specify the transformation from each structure to its associated icon. Because the Object Specification Language should be capable of specifying components at various levels of detail, the resulting visualization should also be capable of displaying various levels of detail.

5.4 Summary

Domain Specific Software Architectures are a natural methodology for using object-oriented requirements analysis, specification, and design; however DSSA's fail to provide the necessary formalism to capture the software architectures information. The development and use of a formal Domain Model and Object Specification Language, along with a formal object base and manipulation system, provide the following capabilities for the above described DSSA.

1. Software component's structural and behavioral characteristics are formally described.
2. System users can easily define and prototype potential new components without the overhead of a full scale development effort.
3. A formal method of selecting and assembling components is provided for rapid development of Ada application systems.

This chapter has shown how a Domain Specific Modeling System can greatly enhance a DSSA's capabilities, and how previously developed visualization technology can be included to make the Domain Modeling System even more easy to use and understand.

VI. Conclusions, and Recommendations

6.1 Introduction

This research has been concerned with three main objectives:

1. The development of a formal graphical specification language which represents the Refine wide-spectrum language.
2. The development and prototype of a generalized mechanism for transforming the Refine wide-spectrum specification language into the previously defined graphical specification language.
3. The extension of the developed formal language visualization technology to any formal language which can be represented in the Refine object base. A Domain Specific Language used in a formalized Domain Specific Software Architecture is an example of such a language.

All three objectives have been achieved. Chapter III systematically produces a formal definition for the Visual Refine graphical specification language. Visual Refine allows for the visualization of a sufficient subset of the Refine formal specification language to be of general use. Chapter IV develops a research prototype visual transformation system which effectively transforms structures in the Refine wide-spectrum specification language into graph-based diagrams using the Visual Refine language developed in Chapter III. And finally, Chapter V provides a methodology for extending the formal language visualization technology to the area of Domain Specific Software Architectures. Specifically, this research has accomplished the following:

- A formal definition for the Visual Refine graphical specification language has been developed.

- The Visual Refine graphical specification language has been implemented as a set of formal transformations; each transformation is encapsulated in a Refine rule, to form a formal, rule based, transformation system.
- A process for identifying and firing applicable Visual Refine rules has been developed. This process facilitates the systematic construction of Visual Refine Diagrams.
- Several enhancements and extensions to the Visual Refine formal transformation system have been identified, along with recommended methodologies for implementing these enhancements and extensions. Extensions and enhancements include:

    1. extension of the Visual Refine language to include Refine control structures, such as an if-then-else.
    2. enhance user friendliness with an automatic diagram generation system.

- A methodology for applying the newly developed formal language visualization technology in the area of Domain Specific Software Architectures has been developed.

6.2 Conclusions

Several conclusions can be made as a result of this research in Graph-Based Visualization. They are:

- Formal transformations from complex, difficult to understand, formal specifications, to simpler, more easily understood, graph-based visualizations is an achievable means of making formal specifications a more widely accepted method of software development. A major hinderance to the use of formal methods is the inherent complexity associated with the formal specification. Graph-Based diagrams offer a more natural, less complex, way of presenting the complicated information contained in a formal specification.

- The Refine formal specification environment is an ideal platform for building a formal graph-based visualization transformation system, such as Visual Refine. The visualization technology developed by this research depends upon being able to recognize and manipulate abstract syntax tree structures represented in an object base. Refine's built-in tools for constructing and manipulating abstract syntax tree structures are extensive. This allows the developer to concentrate on the problem, and its solution, rather than on the mechanics of the development environment.
- Domain Specific Software Architectures can be enhanced, providing greater benefits and more power, by developing a formal Domain Model and Object Specification Language. By incorporating graph-based visualization techniques, developed in this thesis, into the Domain Model and Object Specification Language, complicated software and application structures can be more easily understood.

6.3 Recommendations for Future Research

Several areas of future research can be derived from the research contained in this thesis. Listed below are research areas which could be continued directly from this research.

- Visual Refine is currently uni-directional, meaning that the transformation goes from the formal specification to the graph-based visualization. The formal foundation provided by this research along with the powerful graphical input facilities of the Intervista subsystem of Refine provide a natural mechanism for implementing the inverse transformations.
- Enhance the Visual Refine Transformation System to include non-icon producing transformations. This would give Visual Refine the capability to recognize, and process, abstract syntax tree structures that do not result in the generation of an icon, but rather serve only to link other abstract syntax tree structures

together; the 'rule-impl-op' is one such structure which could be processed to generate the control links necessary to describe the Refine transform construct.

- Further develop the Visual Refine rules, to provide an automated diagram generation system. This requires a careful analysis of each rule to determine how it should effect the object base in preparation for the next rule firing. Additionally, each rule needs to be evaluated as to its relationship with other rules to determine the sequencing used by the methodology described in section 4.6.
- Develop methodologies and algorithms for the automatic layout of icons, and the automatic routing of links, in support of the visualization technologies developed in this thesis.
- Investigate methods of linking the various graph-based diagrams together in a hierarchical manner. This type of methodology would provide the capability to construct a formal specification from a set of diagrams, beginning with a high level concept diagram which can then be further refined in much the same way as Data Flow Diagrams are constructed. Another aspect of this methodology is the capability to automatically generate a visualization of an entire system by constructing the graph hierarchy from the implicit hierarchy of the formal specification.
- A motivating factor for this thesis was to use visual technologies to allow greater understanding of formal specifications; resulting in greater acceptance of formal specifications as a software engineering tool. Furthermore, a substantial increase in overall life cycle productivity, and thus a substantial decrease in overall life cycle costs, is expected as a result of using Balzer's Automation-Based life cycle model. Although it is believed that the visual technologies developed in this thesis have indeed made formal specifications easier to understand, research aimed at quantifying improved understanding, increased life cycle productivity, and decreased life cycle costs, is needed.

- Chapter V only lays the very basics of a foundation for enhancing Domain Specific Software Architectures. This basic methodology needs to be developed into a prototype system which verifies the benefits and potential of a formal Domain Model. This area of research is wide open and could include several independent but related research topics to include:

    1. the construction of a formal Domain Model based upon the Requirements Modeling Language in support of DSSA's in general.
    2. the construction of a formal Domain Model and Object Description Language which supports the specific needs of a formalized DSSA, and
    3. the construction of an overall prototype formal based Domain Modeling System. This overall prototype system would need to be able to support the formal development of software components, the manipulation of software components within the object base, and the assembly of components into a software application.

6.4 Final Remarks

Software Engineering can not, and should not, change overnight; however Software Engineering must change. The current methodologies, even with the advent of computer aided software engineering (CASE) tools, can only offer small improvements to the problems addressed in Chapter I of this thesis. Formal methods of software engineering, on the other hand, potentially offer order of magnitude improvements to the software industry. Improvements which can have as profound an effect on the field of software engineering as the invention of the numeral zero and in-place notation had on the field of mathematics (14). Casual disregard for formal methodologies will make even the most respected software development gurus antiquated relics in the not to distant future.

Appendix A. *Refine^{TM} Primitive Operations Categorized by Operand Data Type*

The following information is directly obtained from (15:appendix D) and is included to aid in the readers understanding of how the Visual Refine formal graphical specification language was developed.

This appendix contains Refine's primitive operations categorized by the operand types of the operations.

- Numbers
  - Addition
  - Subtraction
  - Multiplication
  - Division
  - Integer Division
  - Integer Remainder (Modulo)
  - Integer to Real Coercion
  - Equality
  - Greater Than
  - Greater Than or Equal To
  - Less Than
  - Less Than or Equal To
- Characters
  - Equality
  - Greater Than
  - Greater Than or Equal To
  - Less Than
  - Less Than or Equal To
- Booleans
  - Negation
  - Conjunction
  - Disjunction

  - Implication
  - Ordered Conjunction
  - Ordered Disjunction
  - Universal Quantification
  - Existential Quantification
  - Nondeterministic Choice
  - Equality
- Symbols
  - Symbol To String Coercion
  - Equality
- Sets
  - Size
  - Arbitrary Element
  - Element Addition
  - Element Deletion
  - Union
  - Intersection
  - Set Difference
  - Filter by a Predicate
  - Reduction by an Operation
  - Set to Sequence Coercion
  - Empty
  - Membership
  - Subset
  - Equality
- Sequences
  - Size
  - n-th Element
  - First Element
  - Last Element
  - Subsequence
  - Following Subsequence
  - Assignment of n-th Element
  - Insertion as n-th Element
  - Append an Element
  - Prepend an Element

  - Delete the n-th Element
  - Reverse
  - Image Under a Map
  - Domain
  - Range
  - Concatenate
  - Filter by a Predicate
  - Reduction by an Operation
  - Sequence to Set Coercion
  - Sequence to Map Coercion
  - Empty
  - Membership
  - Equality
- Strings
  - Greater Than
  - Greater Than or Equal To
  - Less Than
  - Less Than or Equal To
- Tuples
  - Field Retrieval
  - Field Assignment
  - Equality
- Maps
  - Size
  - Filter by a Map
  - Image
  - Domain
  - Range
  - Closure
  - Composition
  - Inverse
  - Map to Binary Relation Coercion
  - Empty
  - Equality
- Binary Relations
  - Image

  - Domain
  - Range
  - Closure
  - Composition
  - Transitive Closure
- Objects
  - Creation
  - Destruction
  - Attribute Assignment
  - Attribute Retrieval

Appendix B. *Refine^{TM} Primitive Operations Categorized by Operation Characteristics*

The following information is directly obtained from (15:appendix E) and is included to aid in the readers understanding of how the Visual Refine formal graphical specification language was developed.

This appendix contains a listing of Refine's primitive operations categorized by conceptual similarities. If multiple primitive operations are grouped together under a single operation heading, a listing of the possible operand types follows the operation heading.

- Simple Assignment
  - Numbers
  - Characters
  - Booleans
  - Symbols
  - Sets
  - Sequences
  - Strings
  - Tuples
  - Maps
  - Binary Relations
  - Objects
- Addition
  - Numbers
  - Sets (Element Addition)
  - Sequences (Concatenation)
- Subtraction
  - Numbers
  - Sets (Element Deletion)
  - Sets (Set Difference)
- Multiplication

- Division
- Integer Division
- Integer Remainder (Modulo)
- Integer to Real Coercion
- Equality
  - Numbers
  - Characters
  - Booleans
  - Symbols
  - Sets
  - Sequences
  - Tuples
  - Maps
- Greater Than
  - Numbers
  - Characters
  - Strings
- Greater Than or Equal To
  - Numbers
  - Characters
  - Strings
- Less Than
  - Numbers
  - Characters
  - Strings
- Less Than or Equal To
  - Numbers
  - Characters
  - Strings
- Symbol to String Coercion
- Negation
- Conjunction
- Ordered Conjunction
- Disjunction

- Ordered Disjunction
- Implication
- Universal Quantification
- Existential Quantification
- Size
  - Sets
  - Sequences
  - Maps
- Arbitrary Element
  - Booleans
  - Sets
- Union
- Intersection
- Filter by a Predicate
  - Sets
  - Sequences
  - Maps
- Set to Sequence Coercion
- Subset Test
- Empty Test
  - Sets
  - Sequences
  - Maps
- Membership
- n-th Element
- First Element
- Last Element
- Subsequence
- Rest of Sequence
- Assign n-th Element
- Insert at n-th Position
- Append Element
- Prepend Element

- Delete n-th Element
- Reverse
- Image
  - Sequences
  - Maps
  - Binary Relations
- Domain
  - Sequences
  - Maps
  - Binary Relations
- Range
  - Sequences
  - Maps
  - Binary Relations
- Sequence to Set Coercion
- Map Coercion
  - Sequences
  - Binary Relations
- Field Retrieval
  - Tuples
  - Objects
- Field Assignment
  - Tuples
  - Objects
- Closure
  - Maps
  - Binary Relations
- Composition
  - Maps
  - Binary Relations
- Inverse
- Map to Binary Relation Coercion
- Transitive Coercion

Appendix C. *Place's Graphical Notation*

This appendix contains a complete listing of the graphical notation developed by Place in (15:5-16 - 5-33). The icons are organized in the same manner that place developed them and generally follow Places decomposition of the Refine wide-spectrum language (15:Appendix D) found in Appendix A. Each of the following figures corresponds to a similar figure found in Places work (15).

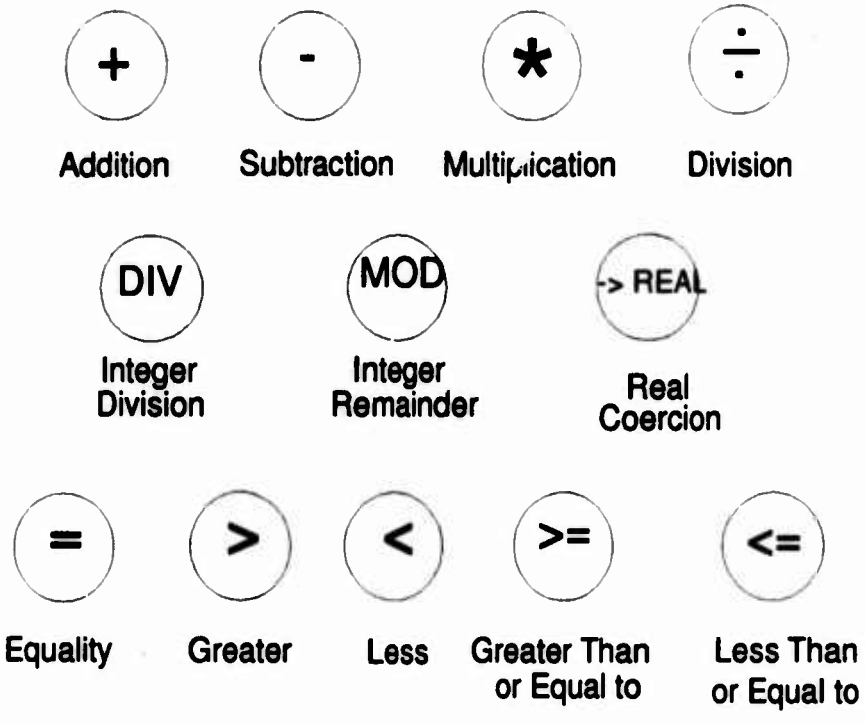
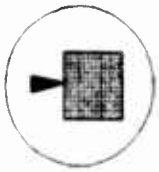


Figure C.1. Mathematical Operations



Simple
Assignment

Figure C.2. Simple Assignment Operator

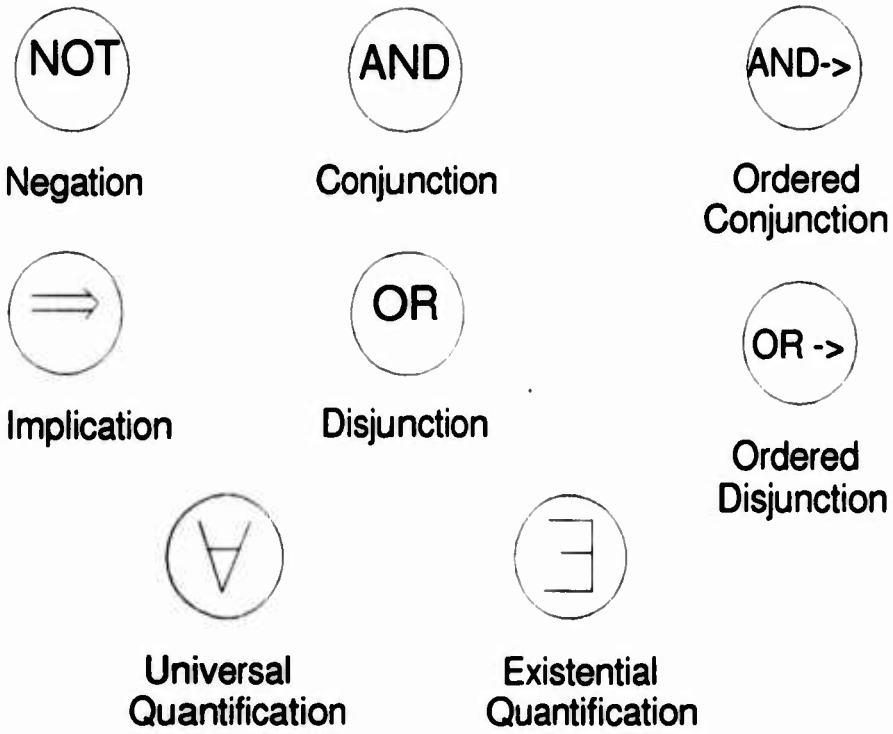


Figure C.3. Boolean Operations

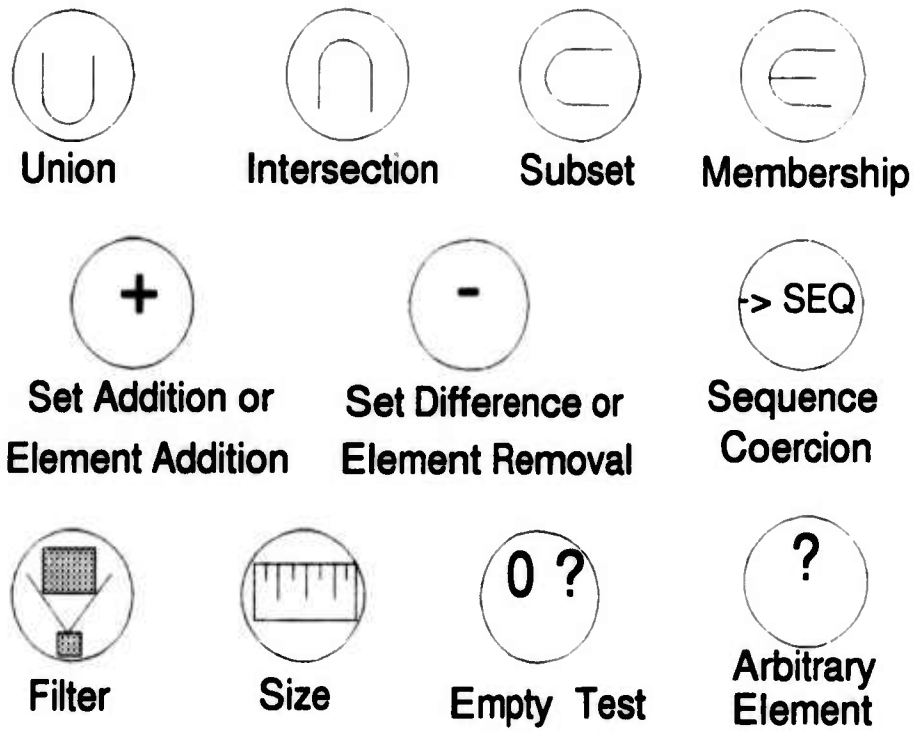


Figure C.4. Set Operations

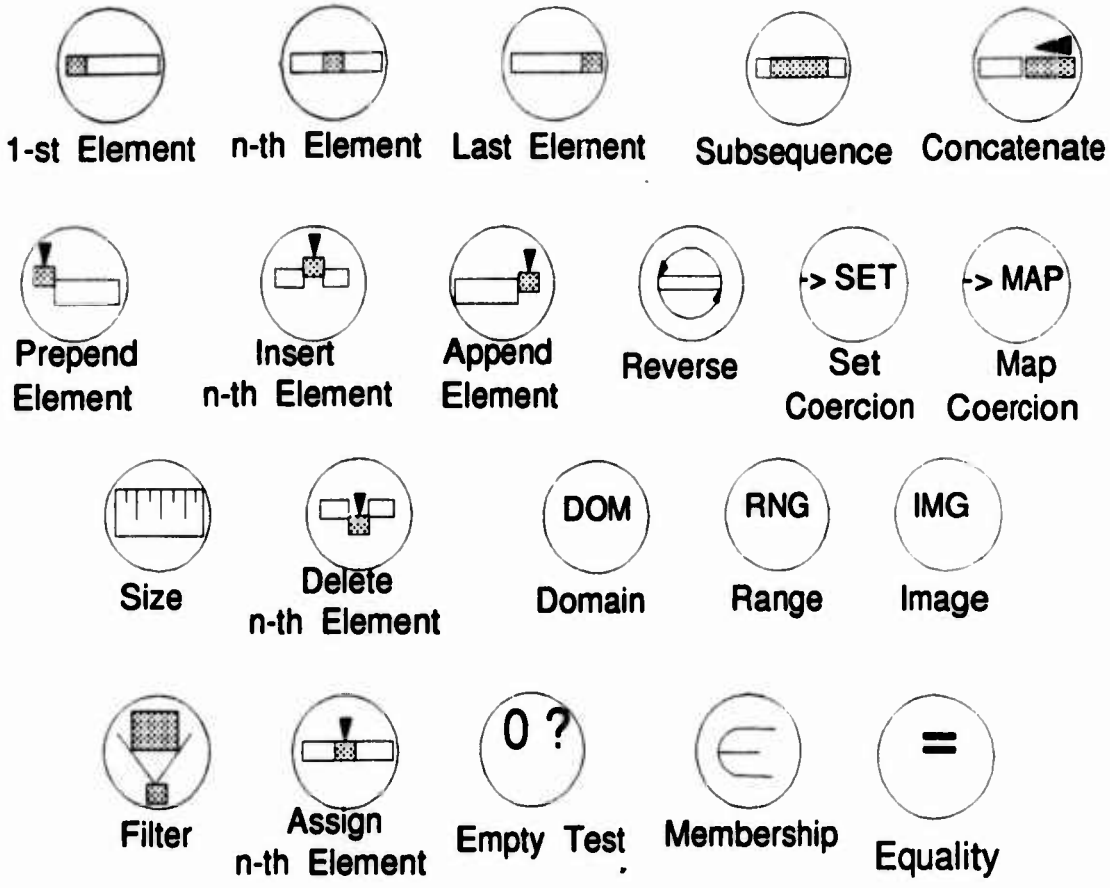


Figure C.5. Sequence Operations

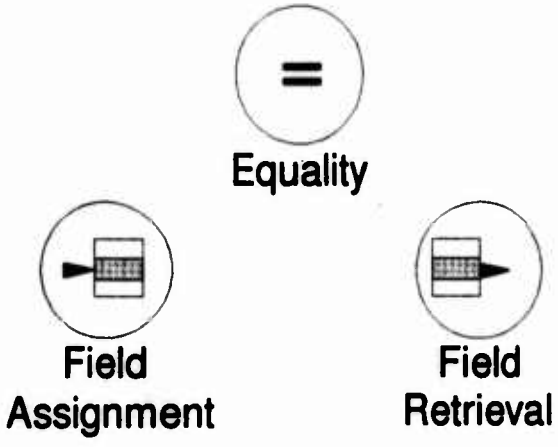


Figure C.6. Tuple Operations

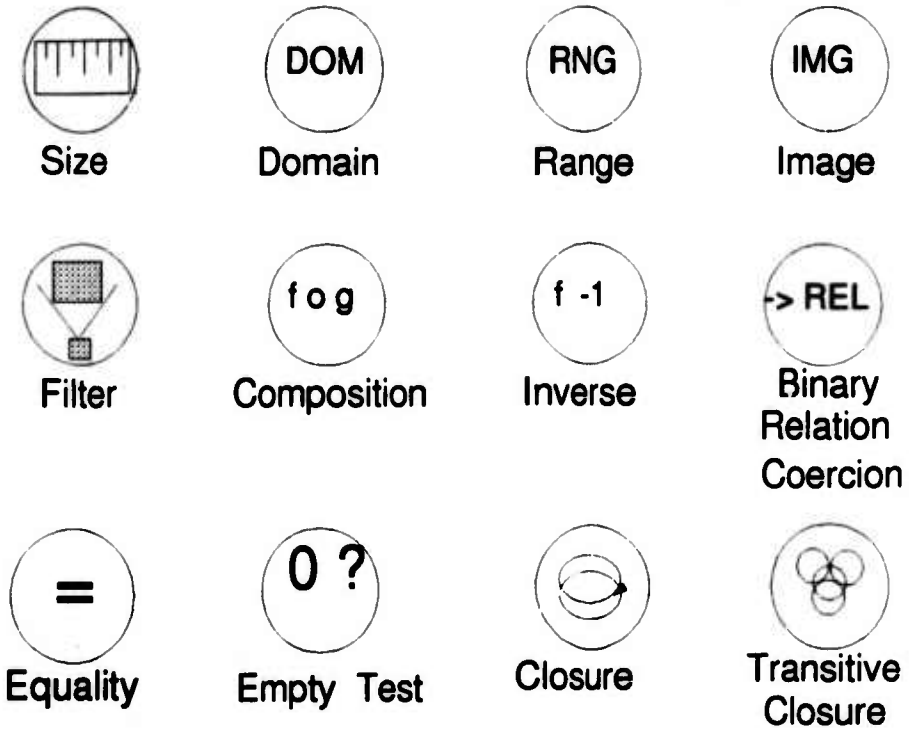


Figure C.7. Map and Binary Relations Operations

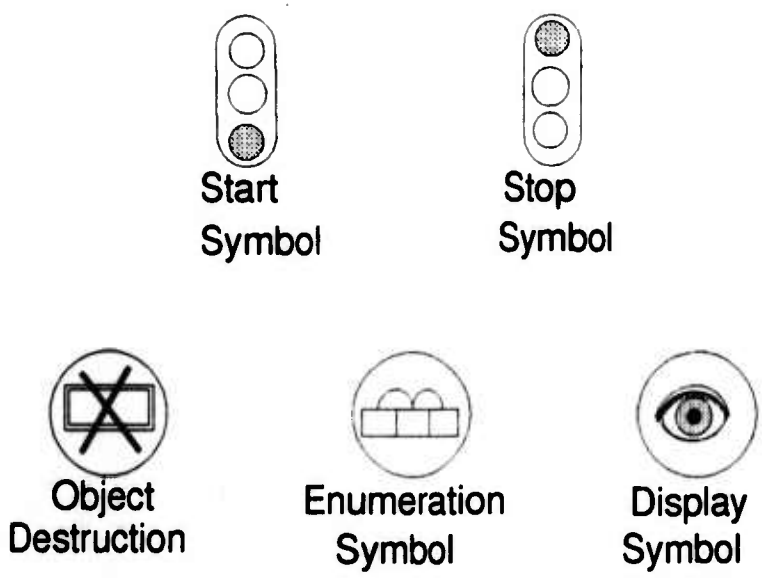


Figure C.8. Miscellaneous Other Operations

Place also includes a notation for distinguishing between an object and other data. Figure C.9 illustrates this distinction.

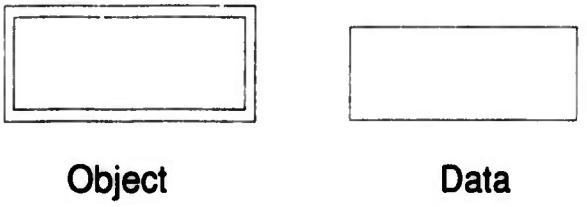


Figure C.9. Data Type Notation

Appendix D. *Refine Abstract Syntax*

This appendix contains a listing of abstract syntax representations of the Refine wide-spectrum specification language. Each icon developed by Place (15:5-16 - 5-33) and listed in Appendix C, is listed here in the internal Refine abstract syntax. The organization of this Appendix is identical to Appendix C, so that matching the abstract syntax to the graphical icon can be done easily. Each of the following figures corresponds directly to a figure found in Appendix C.

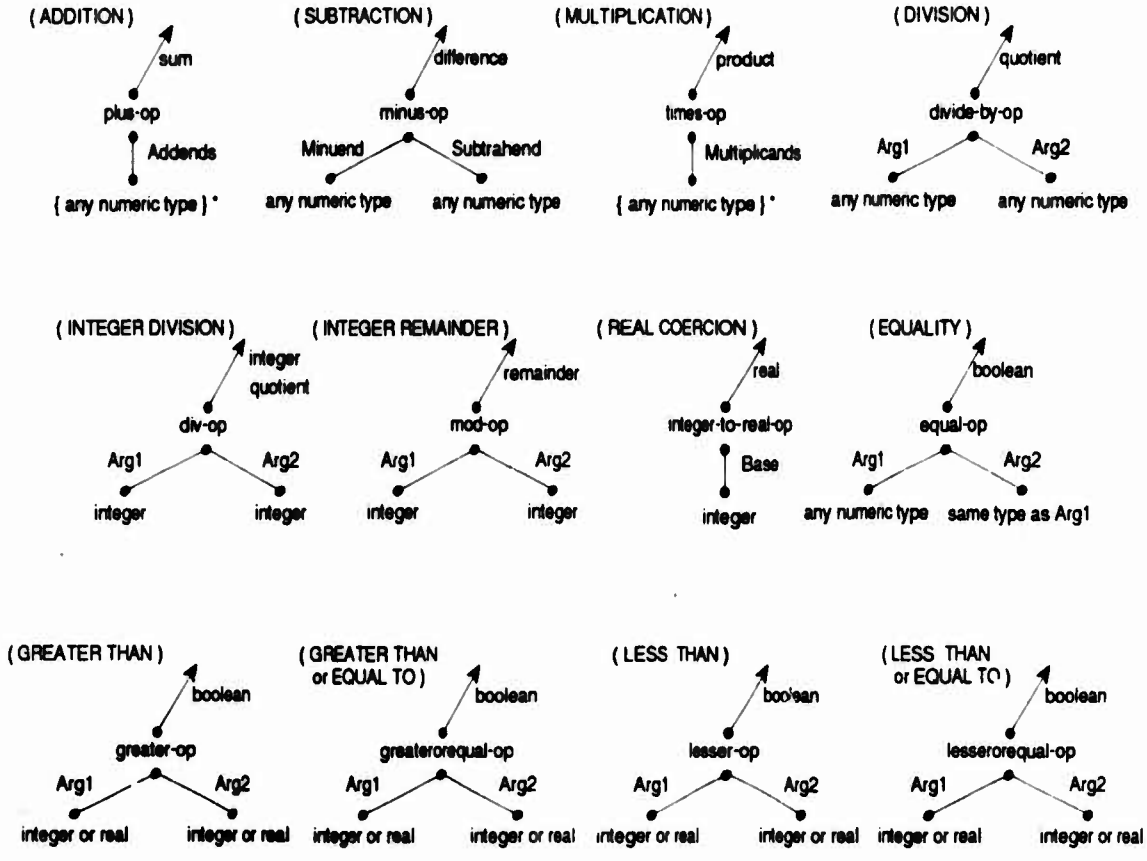


Figure D.1. Mathematical Operations

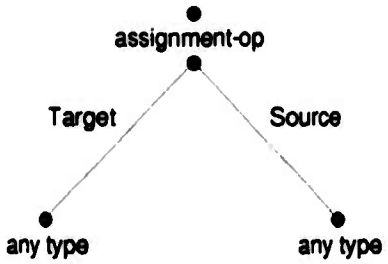


Figure D.2. Simple Assignment Operator

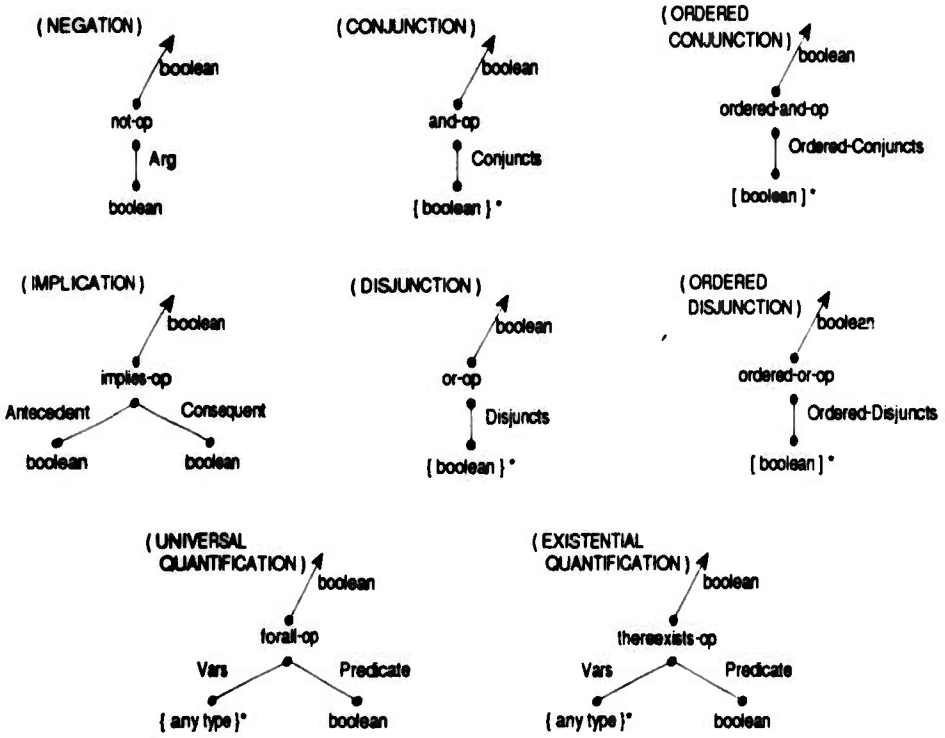


Figure D.3. Boolean Operations

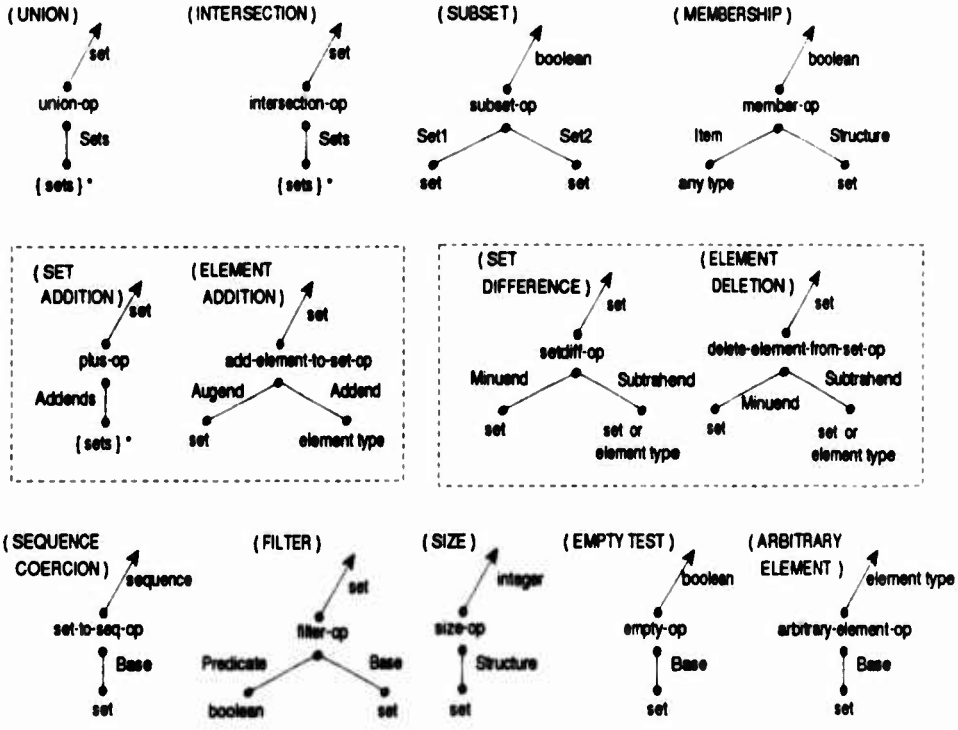


Figure D.4. Set Operations

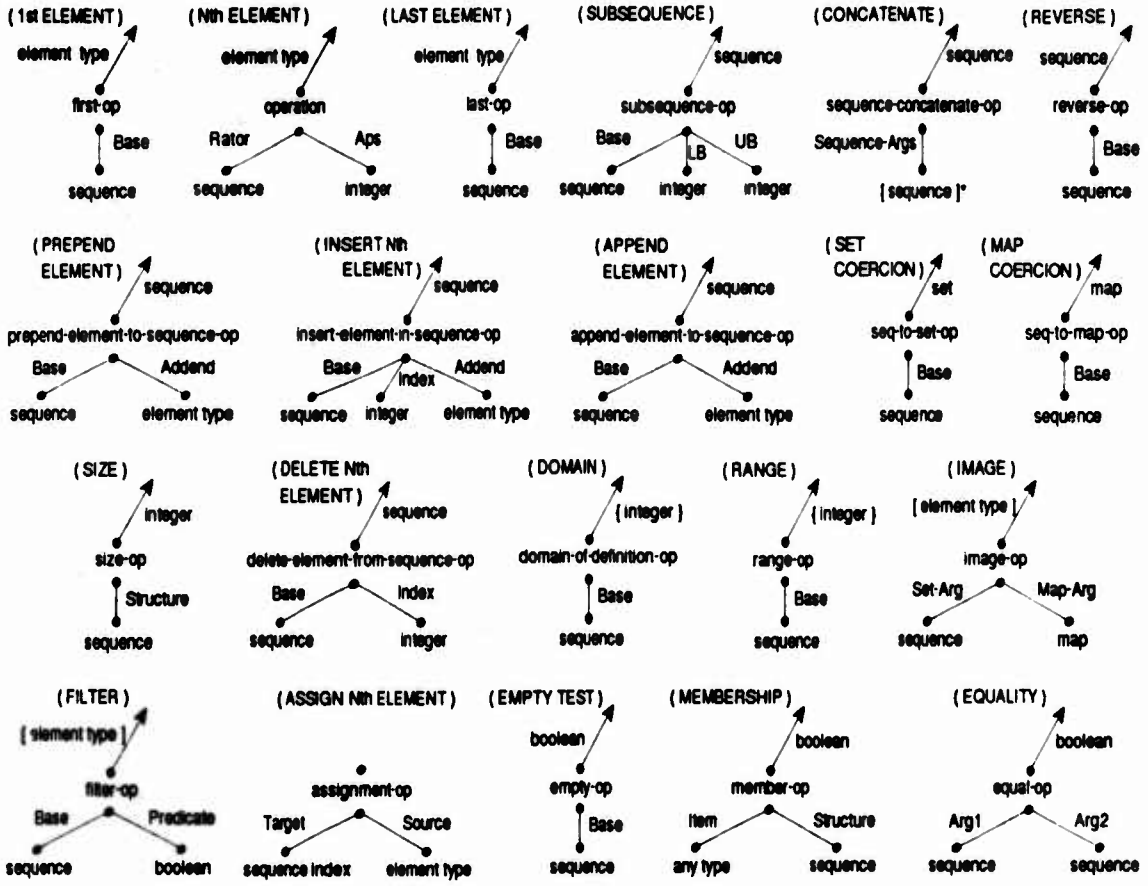


Figure D.5. Sequence Operations

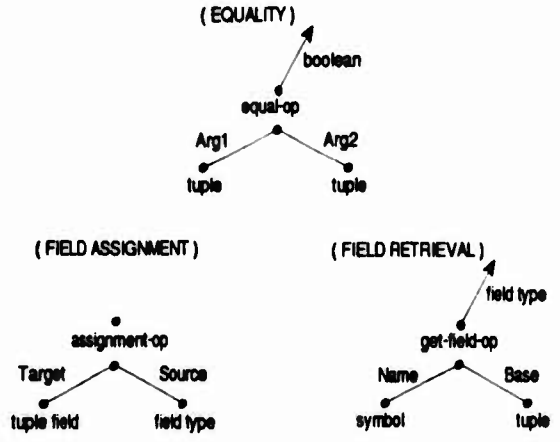


Figure D.6. Tuple Operations

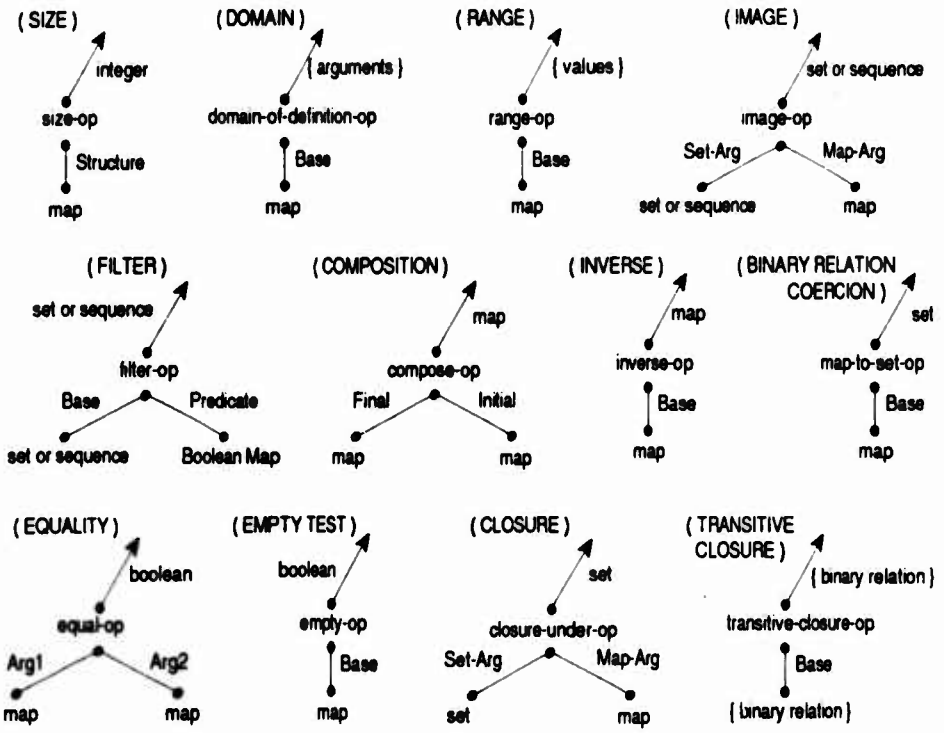


Figure D.7. Map and Binary Relations Operations

The Start, Stop, and Display symbols within Visual Refine are special symbols and have no abstract syntax structure within Refine. There is an alternate "from, to, by" enumeration abstract syntax structure that is not shown. This alternative structure can be rewritten to create the "baseset" structure shown, and therefore the "baseset" enumeration structure has been chosen to represent enumeration.

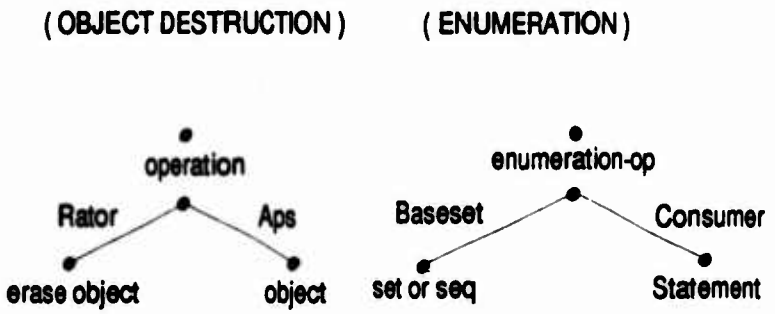


Figure D.8. Miscellaneous Other Operations

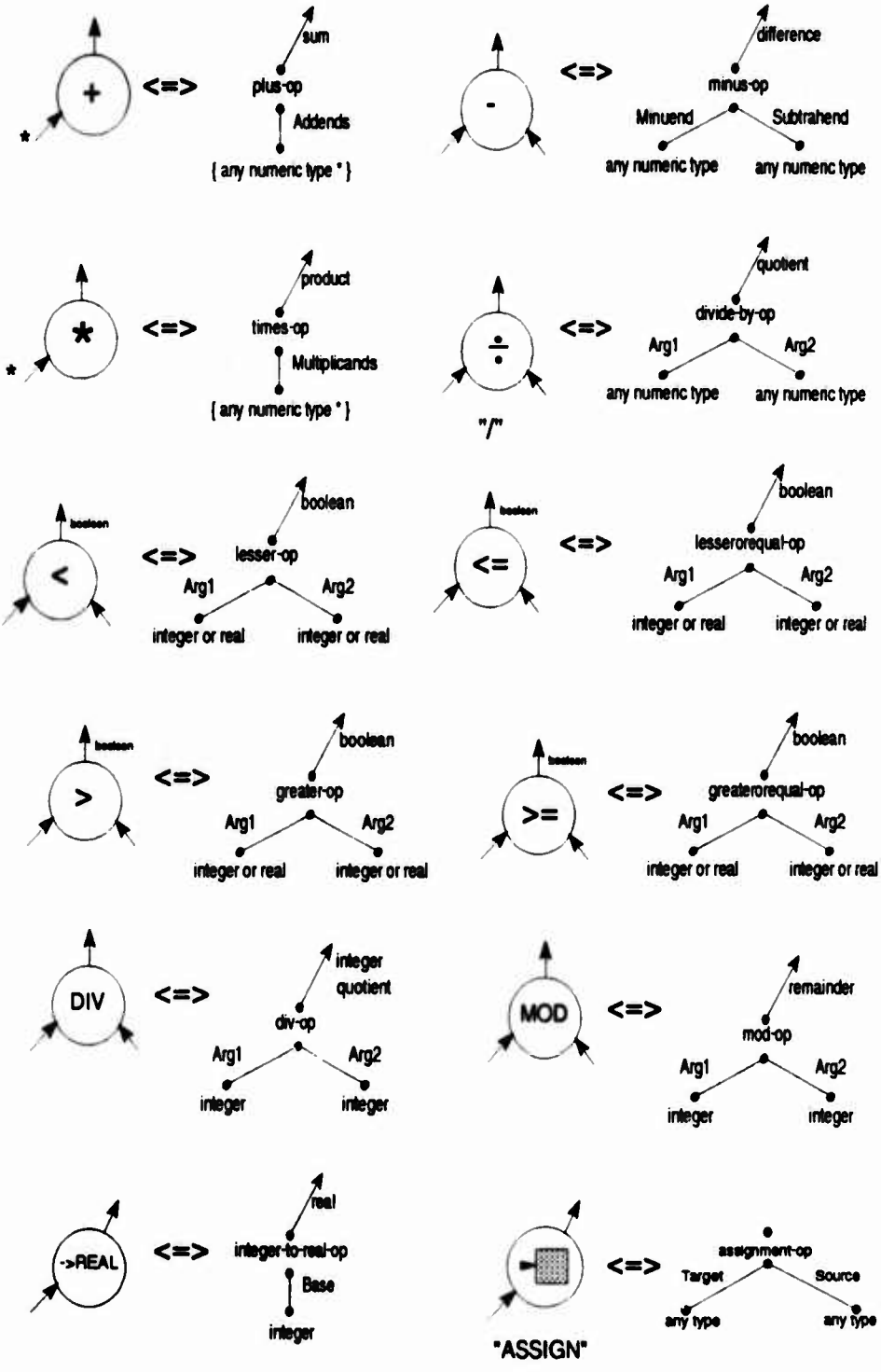
The data notation used by Place (15:5-33) is not an operation, and therefore does not have a specific abstract syntax structure within Refine. The data type icons will be used to represent binding-ref attributes of objects.

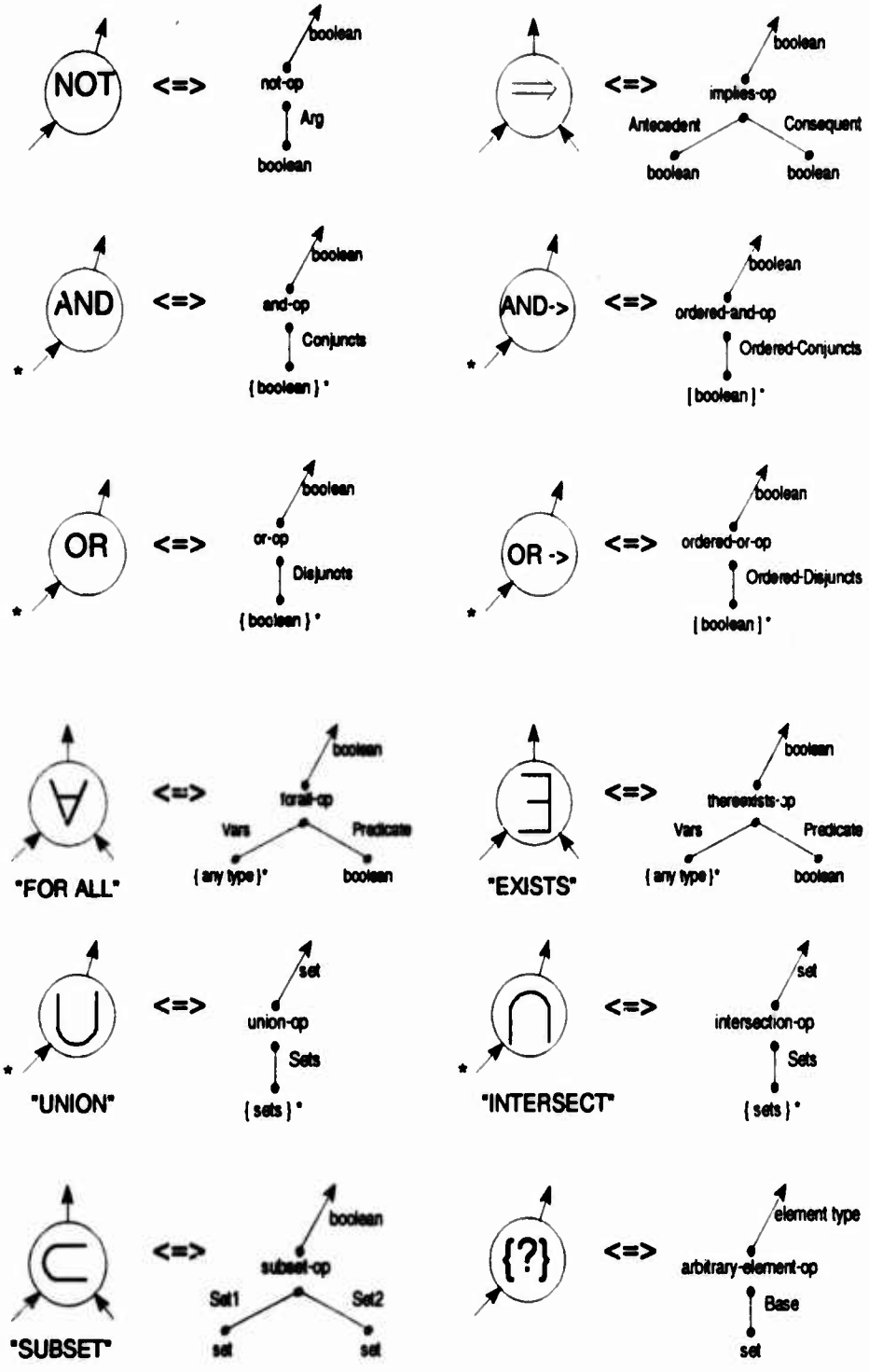
Appendix E. *Visual Refine*

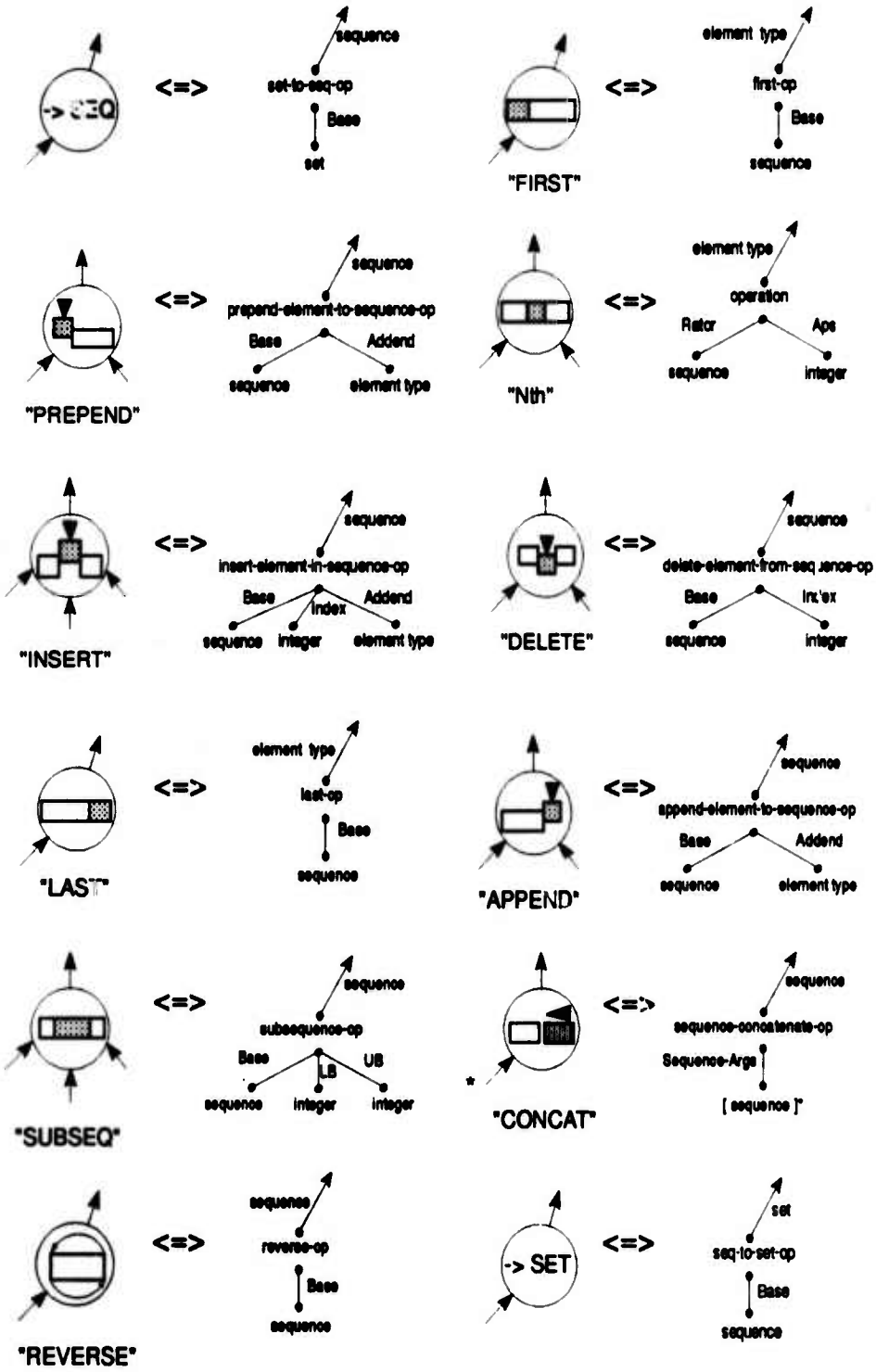
This appendix defines the Visual Refine graphical specification language. The appendix has three sections. The first section contains all icons that were carried over from Place's notation (Appendix C) with only the input and output ports formally defined. The second section contains all icons that were overloaded in Place's notation with the combined Refine abstract syntax tree structure defined. And the last section contains the icons which have been developed as a result of formalizing Place's notation. Also contained in the last section is a list of icons which have been deemed unnecessary or ambiguous to the Visual Refine specification language.

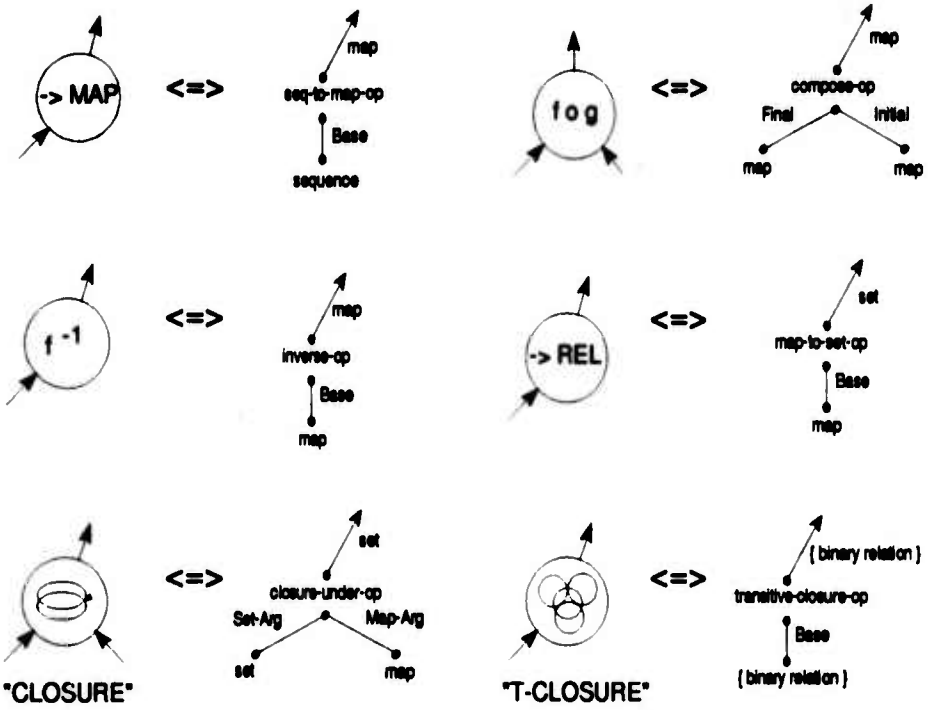
The purpose of this research is to show the applicability of visual technologies toward making formal specifications easier to understand. This research is not necessarily concerned with the implementation issues associated with constructing graphical icons. For this reason, some icons are annotated with a word contained in quotes. This word represents a simpler textual icon, created for the Visual Refine prototype system, developed as part of this research. Textual icons are more easily implemented, as compared to the detailed graphical icons defined for many of the Visual Refine constructs. Therefore, the simpler textual icons have been used; the textual annotation is included in the definition to avoid confusion. Visual Refine icon definitions which do not have the textual annotation have been prototyped as defined.

E.1 Formalized Icons Carried Forward

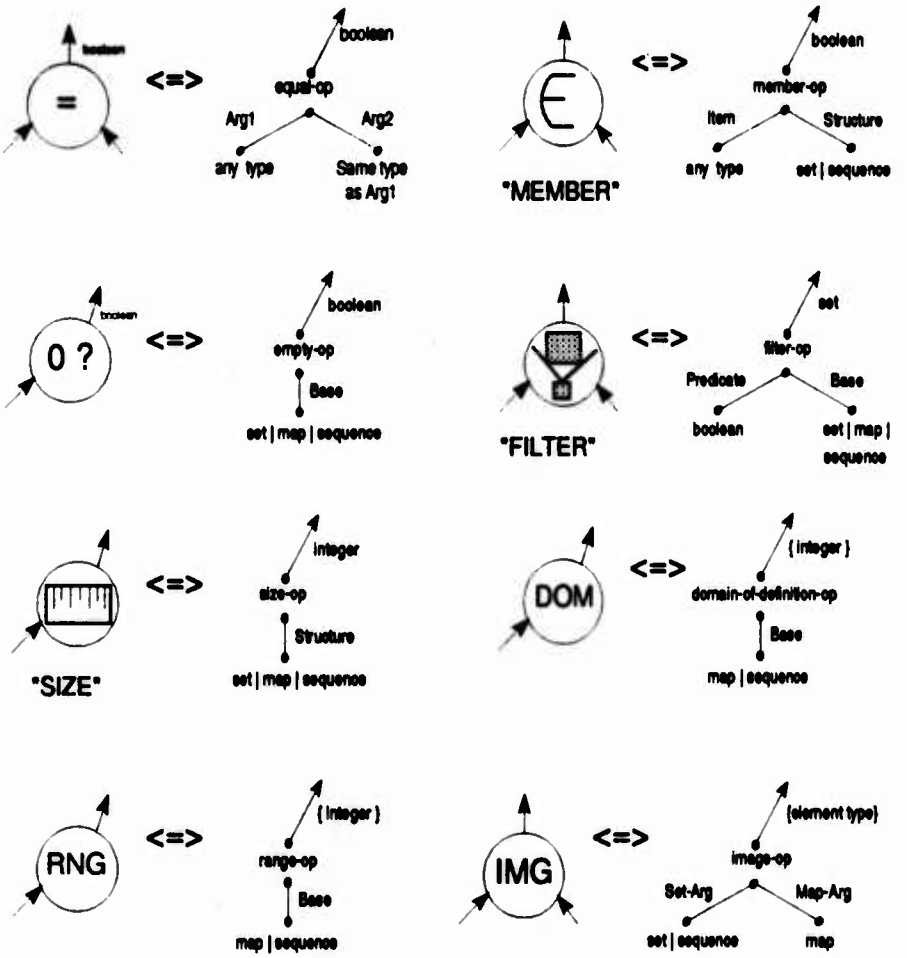



NOT
<=>
boolean
not-op
Arg
boolean
<=>
boolean
implies-op
Antecedent
Consequent
boolean
boolean
AND
<=>
boolean
and-op
Conjuncts
{boolean}*
AND->
<=>
boolean
ordered-and-op
Ordered-Conjuncts
[boolean]*
OR
<=>
boolean
or-op
Disjuncts
{boolean}*
OR ->
<=>
boolean
ordered-or-op
Ordered-Disjuncts
[boolean]*
"FOR ALL"
<=>
boolean
forall-op
Vars
Predicate
{any type}*
boolean
"EXISTS"
<=>
boolean
thereexists-op
Vars
Predicate
{any type}*
boolean
"UNION"
<=>
set
union-op
Sets
{sets}*
"INTERSECT"
<=>
set
intersection-op
Sets
{sets}*
"SUBSET"
<=>
boolean
subset-op
Set1
Set2
set
set
{?}
<=>
element type
arbitrary-element-op
Base
set

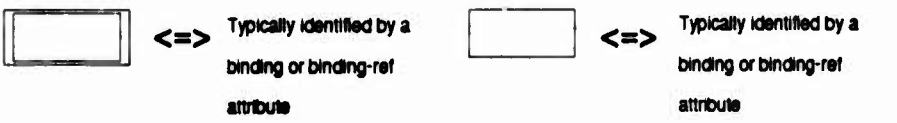

-> SEQ
<=>
sequence
set-to-seq-op
Base
set
"FIRST"
element type
first-op
Base
sequence
"PREPEND"
prepend-element-to-sequence-op
Base
Addend
sequence
element type
"Nth"
operation
Rator
Aps
integer
"INSERT"
insert-element-in-sequence-op
Index
integer
"DELETE"
delete-element-from-sequence-op
"LAST"
last-op
"APPEND"
append-element-to-sequence-op
"SUBSEQ"
subsequence-op
LB
UB
"CONCAT"
sequence-concatenate-op
Sequence-Args
[sequence]*
"REVERSE"
reverse-op
-> SET
set
seq-to-set-op


-> MAP
<=>
map
seq-to-map-op
Base
sequence
f o g
<=>
map
compose-op
Final
Initial
map
map
f -1
<=>
map
inverse-op
Base
map
-> REL
<=>
set
map-to-set-op
Base
map
"CLOSURE"
<=>
set
closure-under-op
Set-Arg
Map-Arg
set
map
"T-CLOSURE"
<=>
{ binary relation }
transitive-closure-op
Base
{ binary relation }
"GET"
<=>
field type
get-field-op
Name
Base
symbol
tuple
"ERASE"
<=>
operation
Rator
Aps
erase object
object
"ENUMERATE"
<=>
enumeration-op
Baseset
Consumer
set or seq
Statement
"DISPLAY"
<=>
operation
Rator
Aps
format
sequence
"START"
<=>
Special Symbol :
No Refine Abstract
Syntax is defined
"STOP"
<=>
Special Symbol :
No Refine Abstract
Syntax is defined

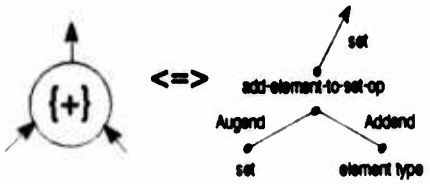
E.2 Overloaded Icons

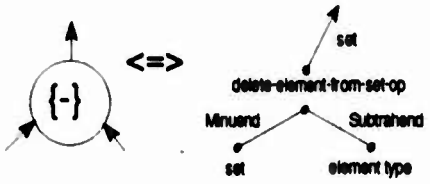


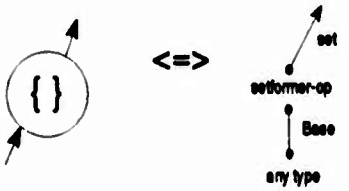
The following two symbols must be evaluated in context with other structures. They are included here for completeness

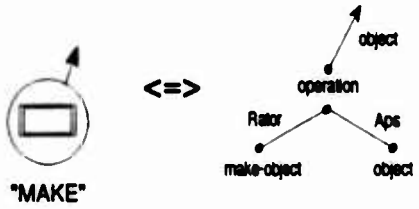


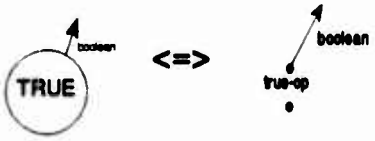
E.3 Newly Developed Icons

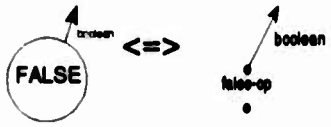


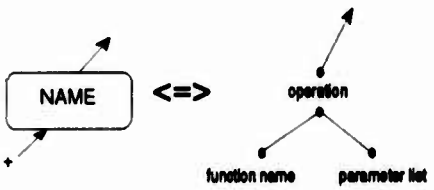




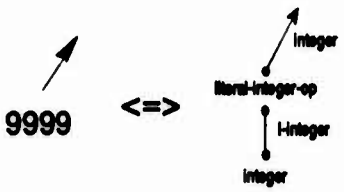




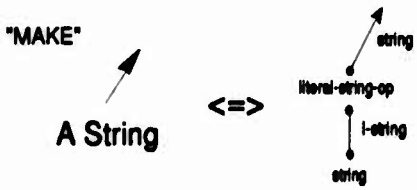




Where 'NAME' is the function name.



Where '9999' is the integer represented by I-integer.



Where 'A String' is the string represented by I-string.

The following icons have not been deemed necessary to the Visual Refine specification language:

- **Assign Nth Element** (sequence operation)
- **Field Assignment** (tuple operation)

Both of the above items are just instances of the simple assignment operation, and therefore are redundant. Including these items will also introduce ambiguity into the Visual Refine specification language.

Appendix F. *Visual Refine Implementation*

This appendix contains the Refine formal specification for the Visual Refine graphical specification language. Each Icon defined in appendix E is implemented using a Refine 'rule' (19:3-167). This appendix is divided into six sections. The first section contains the Visual Refine environment definition, along with some internal utility functions, and the Visual Refine startup rule. Each of the next five sections corrisponds directly to sections, of the same name, in chapter IV.

F.1 Visual Refine Environment

```
!! in-package("RU")
!! in-grammar('user)

% -------- Visual Refine Environment Definitions ----------

var DW-LL-X:      integer = 10
var DW-LL-Y:      integer = 40
var DW-Width:     integer = 550
var DW-Height:    integer = 680

var DW: ri::diagram-window  = undefined
var DS: ri::diagram-surface = undefined

var VR-Icon:            map( Object, ri::icon ) = {||}
var VR-Icon-for-Object: map( Object, ri::icon ) = {||}
var VR-Object-for-Icon: map( ri::icon, Object ) = {||}

form Define-Functional-Converses
   define-fun-converses( 'VR-Icon-for-Object,
                         'VR-Object-for-Icon, true )

var binding-List: set( ri::icon ) = {}

var Stop-Icon: ri::icon = make-object( 'ri::icon )

% ------------ Visual Refine Utility Functions ------------

function In-Binding-List?( L : seq( string ) ) : boolean =
   let( in-list: boolean = false )
      ( enumerate bl: ri::icon over binding-list do
        if( ri::label( bl ) = L ) then
```

```
            in-list <- true
      );
      in-list

function VR-Icon-for-Object?( X: Object ) : boolean =
   if( VR-Icon-for-Object( X ) = undefined ) then
      false
   else
      true


function VR-Icon?( X: Object ) : boolean =
   if( VR-Icon( X ) = undefined ) then
      false
   else
      true


function Get-BL-Icon( L: seq( string ) ) : ri::icon =
   let( found-icon: ri::icon = nil )
      ( enumerate bl: ri::icon over binding-list do
         if( ri::label( bl ) = L ) then
            found-icon <- bl
      );
      found-icon


function VR-Make-String( N: integer ) : string =
   let( int-string: string = [],
        Low-Digit: integer = N,
        Other-Digits: integer = N
      )
      ( if( N ~= 0 ) then
         while( Other-Digits ~= 0 ) do
            Low-Digit <- Other-Digits mod 10;
            Other-Digits <- Other-Digits div 10;
            ( if( Low-Digit = 0 ) then
               int-string <- prepend( int-string, #\0 )
            elseif( Low-Digit = 1 ) then
               int-string <- prepend( int-string, #\1 )
            elseif( Low-Digit = 2 ) then
               int-string <- prepend( int-string, #\2 )
            elseif( Low-Digit = 3 ) then
               int-string <- prepend( int-string, #\3 )
            elseif( Low-Digit = 4 ) then
               int-string <- prepend( int-string, #\4 )
            elseif( Low-Digit = 5 ) then
               int-string <- prepend( int-string, #\5 )
            elseif( Low-Digit = 6 ) then
               int-string <- prepend( int-string, #\6 )
            elseif( Low-Digit = 7 ) then
```

```
            int-string <- prepend( int-string, #\7 )
         elseif( Low-Digit = 8 ) then
            int-string <- prepend( int-string, #\8 )
         elseif( Low-Digit = 9 ) then
            int-string <- prepend( int-string, #\9 )
         )
      else
         int-string <- prepend( int-string, #\0 )
      );
      int-string

% ---------------- Visual Refine Startup ---------------

rule Visual-Refine()
      true
 -->
            binding-list = {}                          &
            DS = make-object( 'ri::diagram-surface )   &
            DW = make-object( 'ri::diagram-window )    &
            ri::surface-viewed( DW ) = DS              &
            ri::window-region ( DW ) =
                  ri::make-region( DW-LL-X, DW-LL-Y,
                                   DW-Width, DW-Height ) &
            ri::window-title( DW ) = "Visual Refine"   &
            ri::window-icon-title( DW ) = "V-Refine"   &
            ri::home-surface( Stop-Icon ) = DS         &
            ri::icon-type( Stop-Icon ) = 'ri::ellipse  &
            ri::height-width-ratio( Stop-Icon ) = 0.5  &
            ri::label( Stop-Icon ) = [ "STOP" ]        &
            ri::clip-icon-label?( Stop-Icon ) = false  &
            ri::position( Stop-Icon ) =
                  ri::make-point(
                        DW-LL-X + (DW-Width - 65 ),
                        DW-LL-Y                    )  &
            ri::expose-window( DW )


% ---------- Visual Refine Control Operations ----------

rule VR-Rule-Op( X: object )
   re::rule-op( X )
 -->
        ri::icon( a ) & ri::home-surface( a ) = DS         &
                        ri::icon-type( a ) = 'ri::ellipse  &
                        ri::height-width-ratio( a ) = 0.5  &
                        ri::label( a ) = [ "START" ]       &
                        ri::clip-icon-label?( a ) = false  &
                        ri::position( a ) =
                           ri::make-point(
                              DW-LL-X + 35,
                              DW-LL-Y + (DW-Height - 100)) &
```

```
                        VR-Object-for-Icon( a ) = X        &
                        VR-Icon-for-Object( X ) = a


rule VR-VFunction-Op( X: object )
   re::vfunction-op( X )
 -->
        ri::icon( a ) & ri::home-surface( a ) = DS         &
                        ri::icon-type( a ) = 'ri::ellipse  &
                        ri::height-width-ratio( a ) = 0.5  &
                        ri::label( a ) = [ "START" ]       &
                        ri::clip-icon-label?( a ) = false  &
                        ri::position( a ) =
                           ri::make-point(
                              DW-LL-X + 35,
                              DW-LL-Y + (DW-Height - 100)) &
                        VR-Object-for-Icon( a ) = X        &
                        VR-Icon-for-Object( X ) = a
```

F.2 Top Level Icons

```
!! in-package("RU")
!! in-grammar('user)


rule VR-Assignment-Op( X: object )
   re::assignment-op( X )
 -->
        ri::icon( a ) & ri::home-surface( a ) = DS         &
                        ri::icon-type( a ) = 'ri::ellipse  &
                        ri::height-width-ratio( a ) = 1.0  &
                        ri::label( a ) = [ "ASSIGN" ]      &
                        ri::clip-icon-label?( a ) = false  &
                        VR-Object-for-Icon( a ) = X


rule VR-Enumerate-Op( X: object )
   re::enumerate-op( X )
 -->
        ri::icon( a ) & ri::home-surface( a ) = DS         &
                        ri::icon-type( a ) = 'ri::ellipse  &
                        ri::height-width-ratio( a ) = 1.0  &
                        ri::label( a ) = [ "ENUMERATE" ]   &
                        ri::clip-icon-label?( a ) = false  &
                        VR-Object-for-Icon( a ) = X        &
                        VR-Icon-for-Object( X ) = a
```

F.3 Embedded Icons

```
!! in-package("RU")
!! in-grammar('user)


rule VR-Literal-Integer-Op( X: object )
   re::literal-integer-Op( X ) &
   VR-Icon-for-Object?( re::parent-expr( X ) )
 -->
        ri::icon( a ) & ri::home-surface( a ) = DS        &
                        ri::icon-type( a ) = 'ri::text    &
                        ri::label( a ) =
                           [ VR-Make-String(
                                re::l-integer( X ) )
                           ]                              &
                        ri::clip-icon-label?( a ) = false &
                        VR-Object-for-Icon( a ) = X       &
        ri::link( b ) & ri::home-surface( b ) = DS        &
                        ri::source( b ) = a               &
                        ri::target( b ) =
                           VR-Icon-for-Object(
                              re::parent-expr( X ))       &
                        ri::dynamic?(b) = true


rule VR-Literal-String-Op( X: object )
   re::literal-string-Op( X ) &
   VR-Icon-for-Object?( re::parent-expr( X ) )
 -->
        ri::icon( a ) & ri::home-surface( a ) = DS        &
                        ri::icon-type( a ) = 'ri::text    &
                        ri::label( a ) =
                           [ re::l-string( X ) ]          &
                        ri::clip-icon-label?( a ) = false &
                        ri::clip-icon-label?( a ) = false &
                        VR-Object-for-Icon( a ) = X       &
        ri::link( b ) & ri::home-surface( b ) = DS        &
                        ri::source( b ) = a               &
                        ri::target( b ) =
                           VR-Icon-for-Object(
                              re::parent-expr( X ))       &
                        ri::dynamic?(b) = true


rule VR-SetFormer-Op( X: object )
```

```
   ( re::setformer-op( X ) or re::literalset-op( X ) ) &
   VR-Icon-for-Object?( re::parent-expr( X ) )
 -->
        ri::icon( a ) & ri::home-surface( a ) = DS        &
                        ri::icon-type( a ) = 'ri::ellipse &
                        ri::height-width-ratio( a ) = 1.0 &
                        ri::label( a ) = [ "SET" ]        &
                        ri::clip-icon-label?( a ) = false &
                        VR-Object-for-Icon( a ) = X       &
                        VR-Icon-for-Object( X ) = a       &
        ri::link( b ) & ri::home-surface( b ) = DS        &
                        ri::source( b ) = a               &
                        ri::target( b ) =
                           VR-Icon-for-Object(
                              re::parent-expr( X ))       &
                        ri::dynamic?(b) = true


rule VR-Plus-Op( X: object )
   re::plus-op( X ) &
   VR-Icon-for-Object?( re::parent-expr( X ) )
 -->
        ri::icon( a ) & ri::home-surface( a ) = DS        &
                        ri::icon-type( a ) = 'ri::ellipse &
                        ri::height-width-ratio( a ) = 1.0 &
                        ri::label( a ) = [ "+" ]          &
                        ri::clip-icon-label?( a ) = false &
                        VR-Object-for-Icon( a ) = X       &
                        VR-Icon-for-Object( X ) = a       &
        ri::link( b ) & ri::home-surface( b ) = DS        &
                        ri::source( b ) = a               &
                        ri::target( b ) =
                           VR-Icon-for-Object(
                              re::parent-expr( X ))       &
                        ri::dynamic?(b) = true


rule VR-Minus-Op( X: object )
   re::minus-op( X ) &
   VR-Icon-for-Object?( re::parent-expr( X ) )
 -->
        ri::icon( a ) & ri::home-surface( a ) = DS        &
                        ri::icon-type( a ) = 'ri::ellipse &
                        ri::height-width-ratio( a ) = 1.0 &
                        ri::label( a ) = [ "-" ]          &
                        ri::clip-icon-label?( a ) = false &
                        VR-Object-for-Icon( a ) = X       &
                        VR-Icon-for-Object( X ) = a       &
        ri::link( b ) & ri::home-surface( b ) = DS        &
                        ri::source( b ) = a               &
```

```
                        ri::target( b ) =
                           VR-Icon-for-Object(
                              re::parent-expr( X ))       &
                        ri::dynamic?(b) = true


rule VR-Times-Op( X: object )
   re::times-op( X ) &
   VR-Icon-for-Object?( re::parent-expr( X ) )
 -->
        ri::icon( a ) & ri::home-surface( a ) = DS        &
                        ri::icon-type( a ) = 'ri::ellipse &
                        ri::height-width-ratio( a ) = 1.0 &
                        ri::label( a ) = [ "*" ]          &
                        ri::clip-icon-label?( a ) = false &
                        VR-Object-for-Icon( a ) = X       &
                        VR-Icon-for-Object( X ) = a       &
        ri::link( b ) & ri::home-surface( b ) = DS        &
                        ri::source( b ) = a               &
                        ri::target( b ) =
                           VR-Icon-for-Object(
                              re::parent-expr( X ))       &
                        ri::dynamic?(b) = true


rule VR-Divided-By-Op( X: object )
   re::divided-by-op( X ) &
   VR-Icon-for-Object?( re::parent-expr( X ) )
 -->
        ri::icon( a ) & ri::home-surface( a ) = DS        &
                        ri::icon-type( a ) = 'ri::ellipse &
                        ri::height-width-ratio( a ) = 1.0 &
                        ri::label( a ) = [ "/" ]          &
                        ri::clip-icon-label?( a ) = false &
                        VR-Object-for-Icon( a ) = X       &
                        VR-Icon-for-Object( X ) = a       &
        ri::link( b ) & ri::home-surface( b ) = DS        &
                        ri::source( b ) = a               &
                        ri::target( b ) =
                           VR-Icon-for-Object(
                              re::parent-expr( X ))       &
                        ri::dynamic?(b) = true


rule VR-Div-Op( X: object )
   re::div-op( X ) &
   VR-Icon-for-Object?( re::parent-expr( X ) )
 -->
        ri::icon( a ) & ri::home-surface( a ) = DS        &
                        ri::icon-type( a ) = 'ri::ellipse &
```

```
                        ri::height-width-ratio( a ) = 1.0 &
                        ri::label( a ) = [ "DIV" ]        &
                        ri::clip-icon-label?( a ) = false &
                        VR-Object-for-Icon( a ) = X       &
                        VR-Icon-for-Object( X ) = a       &
        ri::link( b ) & ri::home-surface( b ) = DS        &
                        ri::source( b ) = a               &
                        ri::target( b ) =
                           VR-Icon-for-Object(
                              re::parent-expr( X ))       &
                        ri::dynamic?(b) = true


rule VR-Mod-Op( X: object )
   re::mod-op( X ) &
   VR-Icon-for-Object?( re::parent-expr( X ) )
 -->
        ri::icon( a ) & ri::home-surface( a ) = DS        &
                        ri::icon-type( a ) = 'ri::ellipse &
                        ri::height-width-ratio( a ) = 1.0 &
                        ri::label( a ) = [ "MOD" ]        &
                        ri::clip-icon-label?( a ) = false &
                        VR-Object-for-Icon( a ) = X       &
                        VR-Icon-for-Object( X ) = a       &
        ri::link( b ) & ri::home-surface( b ) = DS        &
                        ri::source( b ) = a               &
                        ri::target( b ) =
                           VR-Icon-for-Object(
                              re::parent-expr( X ))       &
                        ri::dynamic?(b) = true


rule VR-Integer-To-Real-Op( X: object )
   re::integer-to-real-op( X ) &
   VR-Icon-for-Object?( re::parent-expr( X ) )
 -->
        ri::icon( a ) & ri::home-surface( a ) = DS        &
                        ri::icon-type( a ) = 'ri::ellipse &
                        ri::height-width-ratio( a ) = 1.0 &
                        ri::label( a ) = [ "->REAL" ]     &
                        ri::clip-icon-label?( a ) = false &
                        VR-Object-for-Icon( a ) = X       &
                        VR-Icon-for-Object( X ) = a       &
        ri::link( b ) & ri::home-surface( b ) = DS        &
                        ri::source( b ) = a               &
                        ri::target( b ) =
                           VR-Icon-for-Object(
                              re::parent-expr( X ))       &
                        ri::dynamic?(b) = true
```

```
rule VR-Union-Op( X: object )
   re::union-op( X ) &
   VR-Icon-for-Object?( re::parent-expr( X ) )
 -->
        ri::icon( a ) & ri::home-surface( a ) = DS        &
                        ri::icon-type( a ) = 'ri::ellipse &
                        ri::height-width-ratio( a ) = 1.0 &
                        ri::label( a ) = [ "UNION" ]      &
                        ri::clip-icon-label?( a ) = false &
                        VR-Object-for-Icon( a ) = X       &
                        VR-Icon-for-Object( X ) = a       &
        ri::link( b ) & ri::home-surface( b ) = DS        &
                        ri::source( b ) = a               &
                        ri::target( b ) =
                           VR-Icon-for-Object(
                              re::parent-expr( X ))       &
                        ri::dynamic?(b) = true


rule VR-Intersection-Op( X: object )
   re::intersection-op( X ) &
   VR-Icon-for-Object?( re::parent-expr( X ) )
 -->
        ri::icon( a ) & ri::home-surface( a ) = DS        &
                        ri::icon-type( a ) = 'ri::ellipse &
                        ri::height-width-ratio( a ) = 1.0 &
                        ri::label( a ) = [ "INTERSECT" ]  &
                        ri::clip-icon-label?( a ) = false &
                        VR-Object-for-Icon( a ) = X       &
                        VR-Icon-for-Object( X ) = a       &
        ri::link( b ) & ri::home-surface( b ) = DS        &
                        ri::source( b ) = a               &
                        ri::target( b ) =
                           VR-Icon-for-Object(
                              re::parent-expr( X ))       &
                        ri::dynamic?(b) = true


rule VR-Arbitrary-Element-Op( X: object )
   re::arbitrary-element-op( X ) &
   VR-Icon-for-Object?( re::parent-expr( X ) )
 -->
        ri::icon( a ) & ri::home-surface( a ) = DS        &
                        ri::icon-type( a ) = 'ri::ellipse &
                        ri::height-width-ratio( a ) = 1.0 &
                        ri::label( a ) = [ "{?}" ]        &
                        ri::clip-icon-label?( a ) = false &
                        VR-Object-for-Icon( a ) = X       &
                        VR-Icon-for-Object( X ) = a       &
```

```
        ri::link( b ) & ri::home-surface( b ) = DS        &
                        ri::source( b ) = a               &
                        ri::target( b ) =
                           VR-Icon-for-Object(
                              re::parent-expr( X ))       &
                        ri::dynamic?(b) = true


rule VR-Set-To-Seq-Op( X: object )
   re::set-to-seq-op( X ) &
   VR-Icon-for-Object?( re::parent-expr( X ) )
 -->
        ri::icon( a ) & ri::home-surface( a ) = DS        &
                        ri::icon-type( a ) = 'ri::ellipse &
                        ri::height-width-ratio( a ) = 1.0 &
                        ri::label( a ) = [ "->SEQ" ]      &
                        ri::clip-icon-label?( a ) = false &
                        VR-Object-for-Icon( a ) = X       &
                        VR-Icon-for-Object( X ) = a       &
        ri::link( b ) & ri::home-surface( b ) = DS        &
                        ri::source( b ) = a               &
                        ri::target( b ) =
                           VR-Icon-for-Object(
                              re::parent-expr( X ))       &
                        ri::dynamic?(b) = true


rule VR-First-Op( X: object )
   re::first-op( X ) &
   VR-Icon-for-Object?( re::parent-expr( X ) )
 -->
        ri::icon( a ) & ri::home-surface( a ) = DS        &
                        ri::icon-type( a ) = 'ri::ellipse &
                        ri::height-width-ratio( a ) = 1.0 &
                        ri::label( a ) = [ "FIRST" ]      &
                        ri::clip-icon-label?( a ) = false &
                        VR-Object-for-Icon( a ) = X       &
                        VR-Icon-for-Object( X ) = a       &
        ri::link( b ) & ri::home-surface( b ) = DS        &
                        ri::source( b ) = a               &
                        ri::target( b ) =
                           VR-Icon-for-Object(
                              re::parent-expr( X ))       &
                        ri::dynamic?(b) = true


rule VR-Prepend-Element-To-Sequence-Op( X: object )
   re::prepend-element-to-sequence-op( X ) &
   VR-Icon-for-Object?( re::parent-expr( X ) )

 -->
```

```
        ri::icon( a ) & ri::home-surface( a ) = DS        &
                        ri::icon-type( a ) = 'ri::ellipse &
                        ri::height-width-ratio( a ) = 1.0 &
                        ri::label( a ) = [ "PREPEND" ]    &
                        ri::clip-icon-label?( a ) = false &
                        VR-Object-for-Icon( a ) = X       &
                        VR-Icon-for-Object( X ) = a       &
        ri::link( b ) & ri::home-surface( b ) = DS        &
                        ri::source( b ) = a               &
                        ri::target( b ) =
                           VR-Icon-for-Object(
                              re::parent-expr( X ))       &
                        ri::dynamic?(b) = true


rule VR-Insert-Element-In-Sequence-Op( X: object )
   re::insert-element-in-sequence-op( X ) &
   VR-Icon-for-Object?( re::parent-expr( X ) )
 -->
        ri::icon( a ) & ri::home-surface( a ) = DS        &
                        ri::icon-type( a ) = 'ri::ellipse &
                        ri::height-width-ratio( a ) = 1.0 &
                        ri::label( a ) = [ "INSERT" ]     &
                        ri::clip-icon-label?( a ) = false &
                        VR-Object-for-Icon( a ) = X       &
                        VR-Icon-for-Object( X ) = a       &
        ri::link( b ) & ri::home-surface( b ) = DS        &
                        ri::source( b ) = a               &
                        ri::target( b ) =
                           VR-Icon-for-Object(
                              re::parent-expr( X ))       &
                        ri::dynamic?(b) = true


rule VR-Delete-Element-From-Sequence-Op( X: object )
   re::delete-element-from-sequence-op( X ) &
   VR-Icon-for-Object?( re::parent-expr( X ) )
 -->
        ri::icon( a ) & ri::home-surface( a ) = DS        &
                        ri::icon-type( a ) = 'ri::ellipse &
                        ri::height-width-ratio( a ) = 1.0 &
                        ri::label( a ) = [ "DELETE" ]     &
                        ri::clip-icon-label?( a ) = false &
                        VR-Object-for-Icon( a ) = X       &
                        VR-Icon-for-Object( X ) = a       &
        ri::link( b ) & ri::home-surface( b ) = DS        &
                        ri::source( b ) = a               &
                        ri::target( b ) =
                           VR-Icon-for-Object(
                              re::parent-expr( X ))       &
                        ri::dynamic?(b) = true
```

```
rule VR-Last-Op( X: object )
   re::last-op( X ) &
   VR-Icon-for-Object?( re::parent-expr( X ) )
 -->
        ri::icon( a ) & ri::home-surface( a ) = DS        &
                        ri::icon-type( a ) = 'ri::ellipse &
                        ri::height-width-ratio( a ) = 1.0 &
                        ri::label( a ) = [ "LAST" ]       &
                        ri::clip-icon-label?( a ) = false &
                        VR-Object-for-Icon( a ) = X       &
                        VR-Icon-for-Object( X ) = a       &
        ri::link( b ) & ri::home-surface( b ) = DS        &
                        ri::source( b ) = a               &
                        ri::target( b ) =
                           VR-Icon-for-Object(
                              re::parent-expr( X ))       &
                        ri::dynamic?(b) = true


rule VR-Append-Element-To-Sequence-Op( X: object )
   re::append-element-to-sequence-op( X ) &
   VR-Icon-for-Object?( re::parent-expr( X ) )
 -->
        ri::icon( a ) & ri::home-surface( a ) = DS        &
                        ri::icon-type( a ) = 'ri::ellipse &
                        ri::height-width-ratio( a ) = 1.0 &
                        ri::label( a ) = [ "APPEND" ]     &
                        ri::clip-icon-label?( a ) = false &
                        VR-Object-for-Icon( a ) = X       &
                        VR-Icon-for-Object( X ) = a       &
        ri::link( b ) & ri::home-surface( b ) = DS        &
                        ri::source( b ) = a               &
                        ri::target( b ) =
                           VR-Icon-for-Object(
                              re::parent-expr( X ))       &
                        ri::dynamic?(b) = true


rule VR-Subsequence-Op( X: object )
   re::subsequence-op( X ) &
   VR-Icon-for-Object?( re::parent-expr( X ) )
 -->
        ri::icon( a ) & ri::home-surface( a ) = DS        &
                        ri::icon-type( a ) = 'ri::ellipse &
                        ri::height-width-ratio( a ) = 1.0 &
                        ri::label( a ) = [ "SUBSEQ" ]     &
                        ri::clip-icon-label?( a ) = false &
                        VR-Object-for-Icon( a ) = X       &
                        VR-Icon-for-Object( X ) = a       &
```

```
        ri::link( b ) & ri::home-surface( b ) = DS        &
                        ri::source( b ) = a               &
                        ri::target( b ) =
                           VR-Icon-for-Object(
                              re::parent-expr( X ))       &
                        ri::dynamic?(b) = true


rule VR-Sequence-Concatenate-Op( X: object )
   re::sequence-concatenate-op( X ) &
   VR-Icon-for-Object?( re::parent-expr( X ) )
 -->
        ri::icon( a ) & ri::home-surface( a ) = DS        &
                        ri::icon-type( a ) = 'ri::ellipse &
                        ri::height-width-ratio( a ) = 1.0 &
                        ri::label( a ) = [ "CONCAT" ]     &
                        ri::clip-icon-label?( a ) = false &
                        VR-Object-for-Icon( a ) = X       &
                        VR-Icon-for-Object( X ) = a       &
        ri::link( b ) & ri::home-surface( b ) = DS        &
                        ri::source( b ) = a               &
                        ri::target( b ) =
                           VR-Icon-for-Object(
                              re::parent-expr( X ))       &
                        ri::dynamic?(b) = true


rule VR-Reverse-Op( X: object )
   re::reverse-op( X ) &
   VR-Icon-for-Object?( re::parent-expr( X ) )
 -->
        ri::icon( a ) & ri::home-surface( a ) = DS        &
                        ri::icon-type( a ) = 'ri::ellipse &
                        ri::height-width-ratio( a ) = 1.0 &
                        ri::label( a ) = [ "REVERSE" ]    &
                        ri::clip-icon-label?( a ) = false &
                        VR-Object-for-Icon( a ) = X       &
                        VR-Icon-for-Object( X ) = a       &
        ri::link( b ) & ri::home-surface( b ) = DS        &
                        ri::source( b ) = a               &
                        ri::target( b ) =
                           VR-Icon-for-Object(
                              re::parent-expr( X ))       &
                        ri::dynamic?(b) = true


rule VR-Seq-To-Set-Op( X: object )
   re::seq-to-set-op( X ) &
   VR-Icon-for-Object?( re::parent-expr( X ) )
 -->
        ri::icon( a ) & ri::home-surface( a ) = DS        &
```

```
                        ri::icon-type( a ) = 'ri::ellipse &
                        ri::height-width-ratio( a ) = 1.0 &
                        ri::label( a ) = [ "->SET" ]      &
                        ri::clip-icon-label?( a ) = false &
                        VR-Object-for-Icon( a ) = X       &
                        VR-Icon-for-Object( X ) = a       &
        ri::link( b ) & ri::home-surface( b ) = DS        &
                        ri::source( b ) = a               &
                        ri::target( b ) =
                           VR-Icon-for-Object(
                              re::parent-expr( X ))       &
                        ri::dynamic?(b) = true


rule VR-Seq-To-Map-Op( X: object )
   re::seq-to-map-op( X ) &
   VR-Icon-for-Object?( re::parent-expr( X ) )
 -->
        ri::icon( a ) & ri::home-surface( a ) = DS        &
                        ri::icon-type( a ) = 'ri::ellipse &
                        ri::height-width-ratio( a ) = 1.0 &
                        ri::label( a ) = [ "->MAP" ]      &
                        ri::clip-icon-label?( a ) = false &
                        VR-Object-for-Icon( a ) = X       &
                        VR-Icon-for-Object( X ) = a       &
        ri::link( b ) & ri::home-surface( b ) = DS        &
                        ri::source( b ) = a               &
                        ri::target( b ) =
                           VR-Icon-for-Object(
                              re::parent-expr( X ))       &
                        ri::dynamic?(b) = true


rule VR-Compose-Op( X: object )
   re::compose-op( X ) &
   VR-Icon-for-Object?( re::parent-expr( X ) )
 -->
        ri::icon( a ) & ri::home-surface( a ) = DS        &
                        ri::icon-type( a ) = 'ri::ellipse &
                        ri::height-width-ratio( a ) = 1.0 &
                        ri::label( a ) = [ "f o g" ]      &
                        ri::clip-icon-label?( a ) = false &
                        VR-Object-for-Icon( a ) = X       &
                        VR-Icon-for-Object( X ) = a       &
        ri::link( b ) & ri::home-surface( b ) = DS        &
                        ri::source( b ) = a               &
                        ri::target( b ) =
                           VR-Icon-for-Object(
                              re::parent-expr( X ))       &
                        ri::dynamic?(b) = true
```

```
rule VR-Inverse-Op( X: object )
   re::inverse-op( X ) &
   VR-Icon-for-Object?( re::parent-expr( X ) )
 -->
        ri::icon( a ) & ri::home-surface( a ) = DS        &
                        ri::icon-type( a ) = 'ri::ellipse &
                        ri::height-width-ratio( a ) = 1.0 &
                        ri::label( a ) = [ "f -1" ]       &
                        ri::clip-icon-label?( a ) = false &
                        VR-Object-for-Icon( a ) = X       &
                        VR-Icon-for-Object( X ) = a       &
        ri::link( b ) & ri::home-surface( b ) = DS        &
                        ri::source( b ) = a               &
                        ri::target( b ) =
                           VR-Icon-for-Object(
                              re::parent-expr( X ))       &
                        ri::dynamic?(b) = true


rule VR-Map-To-Set-Op( X: object )
   re::map-to-set-op( X ) &
   VR-Icon-for-Object?( re::parent-expr( X ) )
 -->
        ri::icon( a ) & ri::home-surface( a ) = DS        &
                        ri::icon-type( a ) = 'ri::ellipse &
                        ri::height-width-ratio( a ) = 1.0 &
                        ri::label( a ) = [ "->REL" ]      &
                        ri::clip-icon-label?( a ) = false &
                        VR-Object-for-Icon( a ) = X       &
                        VR-Icon-for-Object( X ) = a       &
        ri::link( b ) & ri::home-surface( b ) = DS        &
                        ri::source( b ) = a               &
                        ri::target( b ) =
                           VR-Icon-for-Object(
                              re::parent-expr( X ))       &
                        ri::dynamic?(b) = true


rule VR-Closure-Under-Op( X: object )
   re::closure-under-op( X ) &
   VR-Icon-for-Object?( re::parent-expr( X ) )
 -->
        ri::icon( a ) & ri::home-surface( a ) = DS        &
                        ri::icon-type( a ) = 'ri::ellipse &
                        ri::height-width-ratio( a ) = 1.0 &
                        ri::label( a ) = [ "CLOSURE" ]    &
                        ri::clip-icon-label?( a ) = false &
                        VR-Object-for-Icon( a ) = X       &
                        VR-Icon-for-Object( X ) = a       &
        ri::link( b ) & ri::home-surface( b ) = DS        &
```

```
                        ri::source( b ) = a                &
                        ri::target( b ) =
                           VR-Icon-for-Object(
                              re::parent-expr( X ))        &
                        ri::dynamic?(b) = true


rule VR-Transitive-Closure-Op( X: object )
   re::transitive-closure-op( X ) &
   VR-Icon-for-Object?( re::parent-expr( X ) )
 -->
        ri::icon( a ) & ri::home-surface( a ) = DS         &
                        ri::icon-type( a ) = 'ri::ellipse  &
                        ri::height-width-ratio( a ) = 1.0  &
                        ri::label( a ) = [ "T-CLOSURE" ]   &
                        ri::clip-icon-label?( a ) = false  &
                        VR-Object-for-Icon( a ) = X        &
                        VR-Icon-for-Object( X ) = a        &
        ri::link( b ) & ri::home-surface( b ) = DS         &
                        ri::source( b ) = a                &
                        ri::target( b ) =
                           VR-Icon-for-Object(
                              re::parent-expr( X ))        &
                        ri::dynamic?(b) = true


rule VR-Get-Field-Op( X: object )
   re::get-field-op( X ) &
   VR-Icon-for-Object?( re::parent-expr( X ) )
 -->
        ri::icon( a ) & ri::home-surface( a ) = DS         &
                        ri::icon-type( a ) = 'ri::ellipse  &
                        ri::height-width-ratio( a ) = 1.0  &
                        ri::label( a ) = [ "GET" ]         &
                        ri::clip-icon-label?( a ) = false  &
                        VR-Object-for-Icon( a ) = X        &
                        VR-Icon-for-Object( X ) = a        &
        ri::link( b ) & ri::home-surface( b ) = DS         &
                        ri::source( b ) = a                &
                        ri::target( b ) =
                           VR-Icon-for-Object(
                              re::parent-expr( X ))        &
                        ri::dynamic?(b) = true


rule VR-Filter-Op( X: object )
   re::filter-op( X ) &
   VR-Icon-for-Object?( re::parent-expr( X ) )
 -->
        ri::icon( a ) & ri::home-surface( a ) = DS         &
                        ri::icon-type( a ) = 'ri::ellipse  &
```

```
                        ri::height-width-ratio( a ) = 1.0 &
                        ri::label( a ) = [ "FILTER" ]     &
                        ri::clip-icon-label?( a ) = false &
                        VR-Object-for-Icon( a ) = X       &
                        VR-Icon-for-Object( X ) = a       &
        ri::link( b ) & ri::home-surface( b ) = DS        &
                        ri::source( b ) = a               &
                        ri::target( b ) =
                           VR-Icon-for-Object(
                              re::parent-expr( X ))       &
                        ri::dynamic?(b) = true


rule VR-Size-Op( X: object )
   re::size-op( X ) &
   VR-Icon-for-Object?( re::parent-expr( X ) )
 -->
        ri::icon( a ) & ri::home-surface( a ) = DS        &
                        ri::icon-type( a ) = 'ri::ellipse &
                        ri::height-width-ratio( a ) = 1.0 &
                        ri::label( a ) = [ "SIZE" ]       &
                        ri::clip-icon-label?( a ) = false &
                        VR-Object-for-Icon( a ) = X       &
                        VR-Icon-for-Object( X ) = a       &
        ri::link( b ) & ri::home-surface( b ) = DS        &
                        ri::source( b ) = a               &
                        ri::target( b ) =
                           VR-Icon-for-Object(
                              re::parent-expr( X ))       &
                        ri::dynamic?(b) = true


rule VR-Domain-Of-Definition-Op( X: object )
   re::domain-of-definition-op( X ) &
   VR-Icon-for-Object?( re::parent-expr( X ) )
 -->
        ri::icon( a ) & ri::home-surface( a ) = DS        &
                        ri::icon-type( a ) = 'ri::ellipse &
                        ri::height-width-ratio( a ) = 1.0 &
                        ri::label( a ) = [ "DOM" ]        &
                        ri::clip-icon-label?( a ) = false &
                        VR-Object-for-Icon( a ) = X       &
                        VR-Icon-for-Object( X ) = a       &
        ri::link( b ) & ri::home-surface( b ) = DS        &
                        ri::source( b ) = a               &
                        ri::target( b ) =
                           VR-Icon-for-Object(
                              re::parent-expr( X ))       &
                        ri::dynamic?(b) = true
```

```
rule VR-Range-Op( X: object )
   re::range-op( X ) &
   VR-Icon-for-Object?( re::parent-expr( X ) )
 -->
        ri::icon( a ) & ri::home-surface( a ) = DS        &
                        ri::icon-type( a ) = 'ri::ellipse &
                        ri::height-width-ratio( a ) = 1.0 &
                        ri::label( a ) = [ "RNG" ]        &
                        ri::clip-icon-label?( a ) = false &
                        VR-Object-for-Icon( a ) = X       &
                        VR-Icon-for-Object( X ) = a       &
        ri::link( b ) & ri::home-surface( b ) = DS        &
                        ri::source( b ) = a               &
                        ri::target( b ) =
                           VR-Icon-for-Object(
                              re::parent-expr( X ))       &
                        ri::dynamic?(b) = true

rule VR-Image-Op( X: object )
   re::image-op( X ) &
   VR-Icon-for-Object?( re::parent-expr( X ) )
 -->
        ri::icon( a ) & ri::home-surface( a ) = DS        &
                        ri::icon-type( a ) = 'ri::ellipse &
                        ri::height-width-ratio( a ) = 1.0 &
                        ri::label( a ) = [ "IMG" ]        &
                        ri::clip-icon-label?( a ) = false &
                        VR-Object-for-Icon( a ) = X       &
                        VR-Icon-for-Object( X ) = a       &
        ri::link( b ) & ri::home-surface( b ) = DS        &
                        ri::source( b ) = a               &
                        ri::target( b ) =
                           VR-Icon-for-Object(
                              re::parent-expr( X ))       &
                        ri::dynamic?(b) = true


rule VR-Add-Element-To-Set-Op( X: object )
   re::add-element-to-set-op( X ) &
   VR-Icon-for-Object?( re::parent-expr( X ) )
 -->
        ri::icon( a ) & ri::home-surface( a ) = DS        &
                        ri::icon-type( a ) = 'ri::ellipse &
                        ri::height-width-ratio( a ) = 1.0 &
                        ri::label( a ) = [ "{+}" ]        &
                        ri::clip-icon-label?( a ) = false &
                        VR-Object-for-Icon( a ) = X       &
                        VR-Icon-for-Object( X ) = a       &
        ri::link( b ) & ri::home-surface( b ) = DS        &
```

```
                        ri::source( b ) = a                &
                        ri::target( b ) =
                           VR-Icon-for-Object(
                              re::parent-expr( X ))        &
                        ri::dynamic?(b) = true


rule VR-Delete-Element-From-Set-Op( X: object )
   re::delete-element-from-set-op( X ) &
   VR-Icon-for-Object?( re::parent-expr( X ) )
 -->
        ri::icon( a ) & ri::home-surface( a ) = DS         &
                        ri::icon-type( a ) = 'ri::ellipse &
                        ri::height-width-ratio( a ) = 1.0 &
                        ri::label( a ) = [ "{-}" ]         &
                        ri::clip-icon-label?( a ) = false &
                        VR-Object-for-Icon( a ) = X        &
                        VR-Icon-for-Object( X ) = a        &
        ri::link( b ) & ri::home-surface( b ) = DS         &
                        ri::source( b ) = a                &
                        ri::target( b ) =
                           VR-Icon-for-Object(
                              re::parent-expr( X ))        &
                        ri::dynamic?(b) = true


rule VR-True-Op( X: object )
   re::true-op( X ) &
   VR-Icon-for-Object?( re::parent-expr( X ) )
 -->
        ri::icon( a ) & ri::home-surface( a ) = DS         &
                        ri::icon-type( a ) = 'ri::ellipse &
                        ri::height-width-ratio( a ) = 1.0 &
                        ri::label( a ) = [ "TRUE" ]        &
                        ri::clip-icon-label?( a ) = false &
                        VR-Object-for-Icon( a ) = X        &
                        VR-Icon-for-Object( X ) = a        &
        ri::link( b ) & ri::home-surface( b ) = DS         &
                        ri::source( b ) = a                &
                        ri::target( b ) =
                           VR-Icon-for-Object(
                              re::parent-expr( X ))        &
                        ri::dynamic?(b) = true


rule VR-False-Op( X: object )
   re::false-op( X ) &
   VR-Icon-for-Object?( re::parent-expr( X ) )
 -->
        ri::icon( a ) & ri::home-surface( a ) = DS         &
                        ri::icon-type( a ) = 'ri::ellipse &
```

```
                        ri::height-width-ratio( a ) = 1.0 &
                        ri::label( a ) = [ "FALSE" ]      &
                        ri::clip-icon-label?( a ) = false &
                        VR-Object-for-Icon( a ) = X       &
                        VR-Icon-for-Object( X ) = a       &
        ri::link( b ) & ri::home-surface( b ) = DS        &
                        ri::source( b ) = a               &
                        ri::target( b ) =
                           VR-Icon-for-Object(
                              re::parent-expr( X ))       &
                        ri::dynamic?(b) = true
```

F.4 Boolean Icons

```
!! in-package("RU")
!! in-grammar('user)


rule VR-Top-Level-Lesser-Op( X: object )
   re::lesser-op( X )
 -->
        ri::icon( a ) & ri::home-surface( a ) = DS        &
                        ri::icon-type( a ) = 'ri::ellipse &
                        ri::height-width-ratio( a ) = 1.0 &
                        ri::label( a ) = [ "<" ]          &
                        ri::clip-icon-label?( a ) = false &
                        VR-Object-for-Icon( a ) = X       &
                        VR-Icon-for-Object( X ) = a


rule VR-Embedded-Lesser-Op( X: object )
   re::lesser-op( X ) &
   VR-Icon-for-Object?( re::parent-expr( X ) )
 -->
        ri::icon( a ) & ri::home-surface( a ) = DS        &
                        ri::icon-type( a ) = 'ri::ellipse &
                        ri::height-width-ratio( a ) = 1.0 &
                        ri::label( a ) = [ "<" ]          &
                        ri::clip-icon-label?( a ) = false &
                        VR-Object-for-Icon( a ) = X       &
                        VR-Icon-for-Object( X ) = a       &
        ri::link( b ) & ri::home-surface( b ) = DS        &
                        ri::source( b ) = a               &
                        ri::target( b ) =
                           VR-Icon-for-Object(
                              re::parent-expr( X ))       &
```

```
                        ri::dynamic?(b) = true


rule VR-Top-Level-LesserOrEqual-Op( X: object )
   re::lesserorequal-op( X )
 -->
        ri::icon( a ) & ri::home-surface( a ) = DS        &
                        ri::icon-type( a ) = 'ri::ellipse &
                        ri::height-width-ratio( a ) = 1.0 &
                        ri::label( a ) = [ "<=" ]         &
                        ri::clip-icon-label?( a ) = false &
                        VR-Object-for-Icon( a ) = X       &
                        VR-Icon-for-Object( X ) = a


rule VR-Embedded-LesserOrEqual-Op( X: object )
   re::lesserorequal-op( X ) &
   VR-Icon-for-Object?( re::parent-expr( X ) )
 -->
        ri::icon( a ) & ri::home-surface( a ) = DS        &
                        ri::icon-type( a ) = 'ri::ellipse &
                        ri::height-width-ratio( a ) = 1.0 &
                        ri::label( a ) = [ "<=" ]         &
                        ri::clip-icon-label?( a ) = false &
                        VR-Object-for-Icon( a ) = X       &
                        VR-Icon-for-Object( X ) = a       &
        ri::link( b ) & ri::home-surface( b ) = DS        &
                        ri::source( b ) = a               &
                        ri::target( b ) =
                           VR-Icon-for-Object(
                              re::parent-expr( X ))       &
                        ri::dynamic?(b) = true


rule VR-Top-Level-Greater-Op( X: object )
   re::greater-op( X )
 -->
        ri::icon( a ) & ri::home-surface( a ) = DS        &
                        ri::icon-type( a ) = 'ri::ellipse &
                        ri::height-width-ratio( a ) = 1.0 &
                        ri::label( a ) = [ ">" ]          &
                        ri::clip-icon-label?( a ) = false &
                        VR-Object-for-Icon( a ) = X       &
                        VR-Icon-for-Object( X ) = a


rule VR-Embedded-Greater-Op( X: object )
   re::greater-op( X ) &
   VR-Icon-for-Object?( re::parent-expr( X ) )
 -->
        ri::icon( a ) & ri::home-surface( a ) = DS        &
```

```
                        ri::icon-type( a ) = 'ri::ellipse &
                        ri::height-width-ratio( a ) = 1.0 &
                        ri::label( a ) = [ ">" ]          &
                        ri::clip-icon-label?( a ) = false &
                        VR-Object-for-Icon( a ) = X       &
                        VR-Icon-for-Object( X ) = a       &
        ri::link( b ) & ri::home-surface( b ) = DS        &
                        ri::source( b ) = a               &
                        ri::target( b ) =
                           VR-Icon-for-Object(
                              re::parent-expr( X ))       &
                        ri::dynamic?(b) = true


rule VR-Top-Level-GreaterOrEqual-Op( X: object )
   re::greaterorequal-op( X )
 -->
        ri::icon( a ) & ri::home-surface( a ) = DS        &
                        ri::icon-type( a ) = 'ri::ellipse &
                        ri::height-width-ratio( a ) = 1.0 &
                        ri::label( a ) = [ ">=" ]         &
                        ri::clip-icon-label?( a ) = false &
                        VR-Object-for-Icon( a ) = X       &
                        VR-Icon-for-Object( X ) = a


rule VR-Embedded-GreaterOrEqual-Op( X: object )
   re::greaterorequal-op( X ) &
   VR-Icon-for-Object?( re::parent-expr( X ) )
 -->
        ri::icon( a ) & ri::home-surface( a ) = DS        &
                        ri::icon-type( a ) = 'ri::ellipse &
                        ri::height-width-ratio( a ) = 1.0 &
                        ri::label( a ) = [ ">=" ]         &
                        ri::clip-icon-label?( a ) = false &
                        VR-Object-for-Icon( a ) = X       &
                        VR-Icon-for-Object( X ) = a       &
        ri::link( b ) & ri::home-surface( b ) = DS        &
                        ri::source( b ) = a               &
                        ri::target( b ) =
                           VR-Icon-for-Object(
                              re::parent-expr( X ))       &
                        ri::dynamic?(b) = true


rule VR-Top-Level-Not-Op( X: object )
   re::not-op( X )
 -->
        ri::icon( a ) & ri::home-surface( a ) = DS        &
                        ri::icon-type( a ) = 'ri::ellipse &
                        ri::height-width-ratio( a ) = 1.0 &
```

```
                        ri::label( a ) = [ "NOT" ]        &
                        ri::clip-icon-label?( a ) = false &
                        VR-Object-for-Icon( a ) = X       &
                        VR-Icon-for-Object( X ) = a


rule VR-Embedded-Not-Op( X: object )
   re::not-op( X ) &
   VR-Icon-for-Object?( re::parent-expr( X ) )
 -->
        ri::icon( a ) & ri::home-surface( a ) = DS        &
                        ri::icon-type( a ) = 'ri::ellipse &
                        ri::height-width-ratio( a ) = 1.0 &
                        ri::label( a ) = [ "NOT" ]        &
                        ri::clip-icon-label?( a ) = false &
                        VR-Object-for-Icon( a ) = X       &
                        VR-Icon-for-Object( X ) = a       &
        ri::link( b ) & ri::home-surface( b ) = DS        &
                        ri::source( b ) = a               &
                        ri::target( b ) =
                           VR-Icon-for-Object(
                              re::parent-expr( X ))       &
                        ri::dynamic?(b) = true


rule VR-Top-Level-Implies-Op( X: object )
   re::implies-op( X )
 -->
        ri::icon( a ) & ri::home-surface( a ) = DS        &
                        ri::icon-type( a ) = 'ri::ellipse &
                        ri::height-width-ratio( a ) = 1.0 &
                        ri::label( a ) = [ "===>" ]       &
                        ri::clip-icon-label?( a ) = false &
                        VR-Object-for-Icon( a ) = X       &
                        VR-Icon-for-Object( X ) = a


rule VR-Embedded-Implies-Op( X: object )
   re::implies-op( X ) &
   VR-Icon-for-Object?( re::parent-expr( X ) )
 -->
        ri::icon( a ) & ri::home-surface( a ) = DS        &
                        ri::icon-type( a ) = 'ri::ellipse &
                        ri::height-width-ratio( a ) = 1.0 &
                        ri::label( a ) = [ "===>" ]       &
                        ri::clip-icon-label?( a ) = false &
                        VR-Object-for-Icon( a ) = X       &
                        VR-Icon-for-Object( X ) = a       &
        ri::link( b ) & ri::home-surface( b ) = DS        &
                        ri::source( b ) = a               &
```

```
                        ri::target( b ) =
                           VR-Icon-for-Object(
                              re::parent-expr( X ))       &
                        ri::dynamic?(b) = true


rule VR-Top-Level-And-Op( X: object )
   re::and-op( X ) &
   ~VR-Icon-for-Object?( re::parent-expr( X ) )
 -->
        ri::icon( a ) & ri::home-surface( a ) = DS        &
                        ri::icon-type( a ) = 'ri::ellipse &
                        ri::height-width-ratio( a ) = 1.0 &
                        ri::label( a ) = [ "AND" ]        &
                        ri::clip-icon-label?( a ) = false &
                        VR-Object-for-Icon( a ) = X       &
                        VR-Icon-for-Object( X ) = a


rule VR-Embedded-And-Op( X: object )
   re::and-op( X ) &
   VR-Icon-for-Object?( re::parent-expr( X ) )
 -->
        ri::icon( a ) & ri::home-surface( a ) = DS        &
                        ri::icon-type( a ) = 'ri::ellipse &
                        ri::height-width-ratio( a ) = 1.0 &
                        ri::label( a ) = [ "AND" ]        &
                        ri::clip-icon-label?( a ) = false &
                        VR-Object-for-Icon( a ) = X       &
                        VR-Icon-for-Object( X ) = a       &
        ri::link( b ) & ri::home-surface( b ) = DS        &
                        ri::source( b ) = a               &
                        ri::target( b ) =
                           VR-Icon-for-Object(
                              re::parent-expr( X ))       &
                        ri::dynamic?(b) = true


rule VR-Top-Level-Ordered-And-Op( X: object )
   re::ordered-and-op( X )
 -->
        ri::icon( a ) & ri::home-surface( a ) = DS        &
                        ri::icon-type( a ) = 'ri::ellipse &
                        ri::height-width-ratio( a ) = 1.0 &
                        ri::label( a ) = [ "AND->" ]      &
                        ri::clip-icon-label?( a ) = false &
                        VR-Object-for-Icon( a ) = X       &
                        VR-Icon-for-Object( X ) = a


rule VR-Embedded-Ordered-And-Op( X: object )
```

```
   re::ordered-and-op( X ) &
   VR-Icon-for-Object?( re::parent-expr( X ) )
  -->
        ri::icon( a ) & ri::home-surface( a ) = DS        &
                        ri::icon-type( a ) = 'ri::ellipse &
                        ri::height-width-ratio( a ) = 1.0 &
                        ri::label( a ) = [ "AND->" ]      &
                        ri::clip-icon-label?( a ) = false &
                        VR-Object-for-Icon( a ) = X       &
                        VR-Icon-for-Object( X ) = a       &
        ri::link( b ) & ri::home-surface( b ) = DS        &
                        ri::source( b ) = a               &
                        ri::target( b ) =
                           VR-Icon-for-Object(
                              re::parent-expr( X ))       &
                        ri::dynamic?(b) = true


rule VR-Top-Level-Or-Op( X: object )
   re::or-op( X )
  -->
        ri::icon( a ) & ri::home-surface( a ) = DS        &
                        ri::icon-type( a ) = 'ri::ellipse &
                        ri::height-width-ratio( a ) = 1.0 &
                        ri::label( a ) = [ "OR" ]         &
                        ri::clip-icon-label?( a ) = false &
                        VR-Object-for-Icon( a ) = X       &
                        VR-Icon-for-Object( X ) = a


rule VR-Embedded-Or-Op( X: object )
   re::or-op( X ) &
   VR-Icon-for-Object?( re::parent-expr( X ) )
  -->
        ri::icon( a ) & ri::home-surface( a ) = DS        &
                        ri::icon-type( a ) = 'ri::ellipse &
                        ri::height-width-ratio( a ) = 1.0 &
                        ri::label( a ) = [ "OR" ]         &
                        ri::clip-icon-label?( a ) = false &
                        VR-Object-for-Icon( a ) = X       &
                        VR-Icon-for-Object( X ) = a       &
        ri::link( b ) & ri::home-surface( b ) = DS        &
                        ri::source( b ) = a               &
                        ri::target( b ) =
                           VR-Icon-for-Object(
                              re::parent-expr( X ))       &
                        ri::dynamic?(b) = true


rule VR-Top-Level-Ordered-Or-Op( X: object )
   re::ordered-or-op( X )
```

```
 -->
        ri::icon( a ) & ri::home-surface( a ) = DS        &
                        ri::icon-type( a ) = 'ri::ellipse &
                        ri::height-width-ratio( a ) = 1.0 &
                        ri::label( a ) = [ "OR->" ]       &
                        ri::clip-icon-label?( a ) = false &
                        VR-Object-for-Icon( a ) = X       &
                        VR-Icon-for-Object( X ) = a


rule VR-Embedded-Ordered-Or-Op( X: object )
   re::ordered-or-op( X ) &
   VR-Icon-for-Object?( re::parent-expr( X ) )
 -->
        ri::icon( a ) & ri::home-surface( a ) = DS        &
                        ri::icon-type( a ) = 'ri::ellipse &
                        ri::height-width-ratio( a ) = 1.0 &
                        ri::label( a ) = [ "OR->" ]       &
                        ri::clip-icon-label?( a ) = false &
                        VR-Object-for-Icon( a ) = X       &
                        VR-Icon-for-Object( X ) = a       &
        ri::link( b ) & ri::home-surface( b ) = DS        &
                        ri::source( b ) = a               &
                        ri::target( b ) =
                           VR-Icon-for-Object(
                              re::parent-expr( X ))       &
                        ri::dynamic?(b) = true


rule VR-Top-Level-ForAll-Op( X: object )
   re::forall-op( X )
 -->
        ri::icon( a ) & ri::home-surface( a ) = DS        &
                        ri::icon-type( a ) = 'ri::ellipse &
                        ri::height-width-ratio( a ) = 1.0 &
                        ri::label( a ) = [ "FOR ALL" ]    &
                        ri::clip-icon-label?( a ) = false &
                        VR-Object-for-Icon( a ) = X       &
                        VR-Icon-for-Object( X ) = a


rule VR-Embedded-ForAll-Op( X: object )
   re::forall-op( X ) &
   VR-Icon-for-Object?( re::parent-expr( X ) )
 -->
        ri::icon( a ) & ri::home-surface( a ) = DS        &
                        ri::icon-type( a ) = 'ri::ellipse &
                        ri::height-width-ratio( a ) = 1.0 &
                        ri::label( a ) = [ "FOR ALL" ]    &
                        ri::clip-icon-label?( a ) = false &
                        VR-Object-for-Icon( a ) = X       &
```

```
                        VR-Icon-for-Object( X ) = a       &
        ri::link( b ) & ri::home-surface( b ) = DS        &
                        ri::source( b ) = a               &
                        ri::target( b ) =
                           VR-Icon-for-Object(
                              re::parent-expr( X ))       &
                        ri::dynamic?(b) = true


rule VR-Top-Level-ThereExists-Op( X: object )
   re::thereexists-op( X )
 -->
        ri::icon( a ) & ri::home-surface( a ) = DS        &
                        ri::icon-type( a ) = 'ri::ellipse &
                        ri::height-width-ratio( a ) = 1.0 &
                        ri::label( a ) = [ "EXISTS" ]     &
                        ri::clip-icon-label?( a ) = false &
                        VR-Object-for-Icon( a ) = X       &
                        VR-Icon-for-Object( X ) = a


rule VR-Embedded-ThereExists-Op( X: object )
   re::thereexists-op( X ) &
   VR-Icon-for-Object?( re::parent-expr( X ) )
 -->
        ri::icon( a ) & ri::home-surface( a ) = DS        &
                        ri::icon-type( a ) = 'ri::ellipse &
                        ri::height-width-ratio( a ) = 1.0 &
                        ri::label( a ) = [ "EXISTS" ]     &
                        ri::clip-icon-label?( a ) = false &
                        VR-Object-for-Icon( a ) = X       &
                        VR-Icon-for-Object( X ) = a       &
        ri::link( b ) & ri::home-surface( b ) = DS        &
                        ri::source( b ) = a               &
                        ri::target( b ) =
                           VR-Icon-for-Object(
                              re::parent-expr( X ))       &
                        ri::dynamic?(b) = true


rule VR-Top-Level-Subset-Op( X: object )
   re::subset-op( X )
 -->
        ri::icon( a ) & ri::home-surface( a ) = DS        &
                        ri::icon-type( a ) = 'ri::ellipse &
                        ri::height-width-ratio( a ) = 1.0 &
                        ri::label( a ) = [ "SUBSET" ]     &
                        ri::clip-icon-label?( a ) = false &
                        VR-Object-for-Icon( a ) = X       &
                        VR-Icon-for-Object( X ) = a
```

```
rule VR-Embedded-Subset-Op( X: object )
   re::subset-op( X ) &
   VR-Icon-for-Object?( re::parent-expr( X ) )
 -->
        ri::icon( a ) & ri::home-surface( a ) = DS        &
                        ri::icon-type( a ) = 'ri::ellipse &
                        ri::height-width-ratio( a ) = 1.0 &
                        ri::label( a ) = [ "SUBSET" ]     &
                        ri::clip-icon-label?( a ) = false &
                        VR-Object-for-Icon( a ) = X       &
                        VR-Icon-for-Object( X ) = a       &
        ri::link( b ) & ri::home-surface( b ) = DS        &
                        ri::source( b ) = a               &
                        ri::target( b ) =
                           VR-Icon-for-Object(
                              re::parent-expr( X ))       &
                        ri::dynamic?(b) = true


rule VR-Top-Level-Equal-Op( X: object )
   re::equal-op( X )
 -->
        ri::icon( a ) & ri::home-surface( a ) = DS        &
                        ri::icon-type( a ) = 'ri::ellipse &
                        ri::height-width-ratio( a ) = 1.0 &
                        ri::label( a ) = [ "=" ]          &
                        ri::clip-icon-label?( a ) = false &
                        VR-Object-for-Icon( a ) = X       &
                        VR-Icon-for-Object( X ) = a


rule VR-Embedded-Equal-Op( X: object )
   re::equal-op( X ) &
   VR-Icon-for-Object?( re::parent-expr( X ) )
 -->
        ri::icon( a ) & ri::home-surface( a ) = DS        &
                        ri::icon-type( a ) = 'ri::ellipse &
                        ri::height-width-ratio( a ) = 1.0 &
                        ri::label( a ) = [ "=" ]          &
                        ri::clip-icon-label?( a ) = false &
                        VR-Object-for-Icon( a ) = X       &
                        VR-Icon-for-Object( X ) = a       &
        ri::link( b ) & ri::home-surface( b ) = DS        &
                        ri::source( b ) = a               &
                        ri::target( b ) =
                           VR-Icon-for-Object(
                              re::parent-expr( X ))       &
                        ri::dynamic?(b) = true
```

```
rule VR-Top-Level-Member-Op( X: object )
   re::member-op( X )
 -->
        ri::icon( a ) & ri::home-surface( a ) = DS        &
                        ri::icon-type( a ) = 'ri::ellipse &
                        ri::height-width-ratio( a ) = 1.0 &
                        ri::label( a ) = [ "MEMBER" ]     &
                        ri::clip-icon-label?( a ) = false &
                        VR-Object-for-Icon( a ) = X       &
                        VR-Icon-for-Object( X ) = a


rule VR-Embedded-Member-Op( X: object )
   re::member-op( X ) &
   VR-Icon-for-Object?( re::parent-expr( X ) )
 -->
        ri::icon( a ) & ri::home-surface( a ) = DS        &
                        ri::icon-type( a ) = 'ri::ellipse &
                        ri::height-width-ratio( a ) = 1.0 &
                        ri::label( a ) = [ "MEMBER" ]     &
                        ri::clip-icon-label?( a ) = false &
                        VR-Object-for-Icon( a ) = X       &
                        VR-Icon-for-Object( X ) = a       &
        ri::link( b ) & ri::home-surface( b ) = DS        &
                        ri::source( b ) = a               &
                        ri::target( b ) =
                           VR-Icon-for-Object(
                              re::parent-expr( X ))       &
                        ri::dynamic?(b) = true


rule VR-Top-Level-Empty-Op( X: object )
   re::empty-op( X )
 -->
        ri::icon( a ) & ri::home-surface( a ) = DS        &
                        ri::icon-type( a ) = 'ri::ellipse &
                        ri::height-width-ratio( a ) = 1.0 &
                        ri::label( a ) = [ "0 ?" ]        &
                        ri::clip-icon-label?( a ) = false &
                        VR-Object-for-Icon( a ) = X       &
                        VR-Icon-for-Object( X ) = a


rule VR-Embedded-Empty-Op( X: object )
   re::empty-op( X ) &
   VR-Icon-for-Object?( re::parent-expr( X ) )
 -->
        ri::icon( a ) & ri::home-surface( a ) = DS        &
                        ri::icon-type( a ) = 'ri::ellipse &
                        ri::height-width-ratio( a ) = 1.0 &
                        ri::label( a ) = [ "0 ?" ]        &
```

```
                        ri::clip-icon-label?( a ) = false &
                        VR-Object-for-Icon( a ) = X       &
                        VR-Icon-for-Object( X ) = a       &
          ri::link( b ) & ri::home-surface( b ) = DS      &
                        ri::source( b ) = a               &
                        ri::target( b ) =
                           VR-Icon-for-Object(
                              re::parent-expr( X ))       &
                        ri::dynamic?(b) = true
```

F.5 Binding Icons

```
!! in-package("RU")
!! in-grammar('user)


rule VR-Binding( X: object )
   re::binding( X ) &
   ~VR-Icon-for-Object?( X ) &
   ~In-Binding-List?(
      [ symbol-to-string( re::name( X )) ]
                  )
 -->
         ri::icon( a ) & ri::home-surface( a ) = DS       &
                        ri::icon-type( a ) = 'ri::box     &
                        ri::height-width-ratio( a ) = 0.5 &
                        ri::label( a ) =
                           [ symbol-to-string(
                                re::name( X ) ) ]         &
                        ri::clip-icon-label?( a ) = false &
                        VR-Object-for-Icon( a ) = X       &
         binding-list = binding-list with a               &
         ri::link( b ) & ri::home-surface( b ) = DS       &
                        ri::source( b ) =
                           VR-Icon-for-Object(
                              re::parent-expr( X ))       &
                        ri::target( b ) = a               &
                        ri::dynamic?(b) = true



rule VR-Object( X: object )
   re::operation( X ) &
   re::bindingname( re::data-type( re::ref-to(
                     re::rator( X )))) = 'OBJECT-CLASS
 -->
```

```
        ri::icon( a ) & ri::home-surface( a ) = DS        &
                        ri::icon-type( a ) = 'ri::box     &
                        ri::height-width-ratio( a ) = 0.5 &
                        ri::highlight-style( a ) =
                           'ri::reverse-video             &
                        ri::label( a ) =
                           [ symbol-to-string(
                                re::bindingname(
                                   re::rator( X ))) ]     &
                        ri::clip-icon-label?( a ) = false &
                        VR-Object-for-Icon( a ) = X       &
        binding-list = binding-list with a                &
        ri::link( b ) & ri::home-surface( b ) = DS        &
                        ri::source( b ) = a               &
                        ri::target( b ) =
                           VR-Icon-for-Object(
                              re::parent-expr( X ))       &
                        ri::dynamic?(b) = true


rule VR-Data-With-Object-Link( X: object )
   re::operation( X ) &
   In-Binding-List?(
      [ symbol-to-string( re::bindingname(
           first( re::aps( X )))) ]
                ) &
   re::powermapping-op(
      re::data-type( re::ref-to( re::rator( X ) ) )
                   )
 --> .
        ri::icon( a ) & ri::home-surface( a ) = DS        &
                        ri::icon-type( a ) = 'ri::box     &
                        ri::height-width-ratio( a ) = 0.5 &
                         ri::label( a ) =
                           [ symbol-to-string(
                                re::bindingname(
                                   re::rator( X ))) ]     &
                        ri::clip-icon-label?( a ) = false &
                        VR-Object-for-Icon( a ) = X       &
        binding-list = binding-list with a                &
        ri::link( b ) & ri::home-surface( b ) = DS        &
                        ri::source( b ) =
                           VR-Icon-for-Object(
                              re::parent-expr( X ))       &
                        ri::target( b ) = a               &
                        ri::dynamic?(b) = true            &
        ri::link( c ) & ri::home-surface( c ) = DS        &
                        ri::source( c ) = a               &
                        ri::target( c ) = Get-BL-Icon(
                           [ symbol-to-string(
                                re::bindingname(
```

```
                                        first( re::aps( X )))) ]
                                                              )    &
                              ri::dynamic?( c ) = true


rule VR-Data-Without-Object-Link( X: object )
   re::operation( X ) &
   re::powermapping-op( re::data-type( re::ref-to(
                                        re::rator( X ))))
 -->
        ri::icon( a ) & ri::home-surface( a ) = DS        &
                        ri::icon-type( a ) = 'ri::box     &
                        ri::height-width-ratio( a ) = 0.5 &
                         ri::label( a ) =
                           [ symbol-to-string(
                                re::bindingname(
                                   re::rator( X ))) ]     &
                        ri::clip-icon-label?( a ) = false &
                        VR-Object-for-Icon( a ) = X       &
        binding-list = binding-list with a                &
        ri::link( b ) & ri::home-surface( b ) = DS        &
                        ri::source( b ) = a               &
                        ri::target( b ) =
                           VR-Icon-for-Object(
                              re::parent-expr( X ))       &
                        ri::dynamic?(b) = true


rule VR-Binding-Ref-Build-Icon( X: object )
   re::binding-ref( X ) &
   ~re::setformer-op( re::parent-expr( X ) ) &
   ~In-Binding-List?(
      [ symbol-to-string( re::bindingname( X ) ) ] )
 -->
        ri::icon( a ) & ri::home-surface( a ) = DS        &
                        ri::icon-type( a ) = 'ri::box     &
                        ri::height-width-ratio( a ) = 0.5 &
                        ri::label( a ) =
                           [ symbol-to-string(
                                re::bindingname( X )) ]   &
                        ri::clip-icon-label?( a ) = false &
                        VR-Object-for-Icon( a ) = X       &
        binding-list = binding-list with a                &
        ri::link( b ) & ri::home-surface( b ) = DS        &
                        ri::source( b ) = a               &
                        ri::target( b ) =
                           VR-Icon-for-Object(
                              re::parent-expr( X ))       &
                        ri::dynamic?(b) = true
```

```
rule VR-Binding-Ref-Link-Icon( X: object )
   re::binding-ref( X ) &
   re::setformer-op( re::parent-expr( X ) ) &
   In-Binding-List?( [ symbol-to-string(
                          re::bindingname( X ) ) ] )
 -->
        ri::link( b ) & ri::home-surface( b ) = DS        &
                        ri::source( b ) =
                           Get-BL-Icon(
                              [ symbol-to-string(
                                   re::bindingname( X )) ]
                                 )                        &
                        ri::target( b ) =
                              VR-Icon-for-Object(
                                 re::parent-expr( X ))    &
                        ri::dynamic?(b) = true
```

F.6 Miscellaneous Icons

```
!! in-package("RU")
!! in-grammar('user)


% ========================================================
%     The following rules are all represented by the
% Refine 'operation' operator.  Therefore, the precondition
% needs further restrictions, in order to determine when
% each rule should be applied.
% ========================================================

rule VR-Nth-Element-Op( X: object )
   re::operation( X ) &
   In-Binding-List?(
      [ symbol-to-string( re::bindingname( re::rator( X ))) ]
                  ) &
   VR-Icon-for-Object?( re::parent-expr( X ) )
 -->
        ri::icon( a ) & ri::home-surface( a ) = DS        &
                        ri::icon-type( a ) = 'ri::ellipse &
                        ri::height-width-ratio( a ) = 1.0 &
                        ri::label( a ) = [ "Nth" ]        &
                        ri::clip-icon-label?( a ) = false &
                        VR-Object-for-Icon( a ) = X       &
                        VR-Icon-for-Object( X ) = a       &
        ri::link( b ) & ri::home-surface( b ) = DS        &
                        ri::source( b ) = a               &
```

```
                        ri::target( b ) =
                           VR-Icon-for-Object(
                              re::parent-expr( X ))       &
                        ri::dynamic?(b) = true


rule VR-Top-Level-Function-Call( X: object )
   re::operation( X )
 -->
        ri::icon( a ) & ri::home-surface( a ) = DS        &
                        ri::icon-type( a ) = 'ri::diamond &
                        ri::height-width-ratio( a ) = 0.5 &
                        ri::label( a ) = [
                           symbol-to-string( re::bindingname(
                              re::rator( X ))) ]          &
                        ri::clip-icon-label?( a ) = false &
                        VR-Object-for-Icon( a ) = X


rule VR-Embeded-Function-Call( X: object )
   re::operation( X )
 -->
        ri::icon( a ) & ri::home-surface( a ) = DS        &
                        ri::icon-type( a ) = 'ri::diamond &
                        ri::height-width-ratio( a ) = 0.5 &
                        ri::label( a ) = [
                           symbol-to-string( re::bindingname(
                              re::rator( X ))) ]          &
                        ri::clip-icon-label?( a ) = false &
                        VR-Object-for-Icon( a ) = X       &
        ri::link( b ) & ri::home-surface( b ) = DS        &
                        ri::source( b ) = a               &
                        ri::target( b ) =
                           VR-Icon-for-Object(
                              re::parent-expr( X ))       &
                         ri::dynamic?(b) = true


rule VR-Erase-Object-Op( X: object )
   re::operation( X ) &
   re::bindingname( re::rator( X ) ) = 'erase-object
 -->
        ri::icon( a ) & ri::home-surface( a ) = DS        &
                        ri::icon-type( a ) = 'ri::ellipse &
                        ri::height-width-ratio( a ) = 1.0 &
                        ri::label( a ) = [ "ERASE" ]      &
                        ri::clip-icon-label?( a ) = false &
                        VR-Object-for-Icon( a ) = X       &
                        VR-Icon-for-Object( X ) = a
```

```
rule VR-Make-Object-Op( X: object )
   re::operation( X ) &
   re::bindingname( re::rator( X ) ) = 'make-object
 -->
        ri::icon( a ) & ri::home-surface( a ) = DS        &
                        ri::icon-type( a ) = 'ri::ellipse &
                        ri::height-width-ratio( a ) = 1.0 &
                        ri::label( a ) = [ "MAKE" ] &
                        ri::clip-icon-label?( a ) = false &
                        VR-Object-for-Icon( a ) = X       &
                        VR-Icon-for-Object( X ) = a       &
        ri::highlight-style( VR-Icon-for-Object(
                                re::parent-expr( X )) ) =
                                      'ri::reverse-video &
        ri::link( b ) & ri::home-surface( b ) = DS        &
                        ri::source( b ) = a               &
                        ri::target( b ) =
                           VR-Icon-for-Object(
                              re::parent-expr( X ))       &
                        ri::dynamic?(b) = true


rule VR-Display-Object-Op( X: object )
   re::bindingname( re::rator( X ) ) = 'format &
   re::operation( X )
 -->
        ri::icon( a ) & ri::home-surface( a ) = DS        &
                        ri::icon-type( a ) = 'ri::ellipse &
                        ri::height-width-ratio( a ) = 1.0 &
                        ri::label( a ) = [ "DISPLAY" ]    &
                        ri::clip-icon-label?( a ) = false &
                        VR-Object-for-Icon( a ) = X       &
                        VR-Icon-for-Object( X ) = a
```

Appendix G. *Place's Refine^{TM} Specification for Kemmerer's Library Problem*

This appendix contains a Refine specification for Kemmerer's library problem discussed in Chapter IV. This specification is a slightly modified version of Place's Refine specification found in (15:Appendix F).

```
!! in-package("RU")
!! in-grammar('user)

var LIBRARY-WORLD-OBJECT: object-class subtype-of user-object

var BOOK: object-class subtype-of library-world-object
var USER: object-class subtype-of library-world-object

form DECLARE-LIBRARY-UNIQUE-NAMES-CLASSES
   unique-names-class ( 'book, true ) &
   unique-names-class ( 'user, true )

% ------ Define the attributes of a book -------
var BOOK-OUT:            map( book, boolean )       = {||}
var ON-SHELF:            map( book, boolean )       = {||}
var TITLE-OF-BOOK:       map( book, string )        = {||}
var AUTHOR-OF-BOOK:      map( book, string )        = {||}
var SUBJECT-OF-BOOK:     map( book, set( string ) ) = {||}
var LAST-CHECKED-OUT-BY: map( book, string )        = {||}

% ------- Define the attributes of a user -------
var CUSTOMER:  map( user, boolean ) = {||}
var STAFF:     map( user, boolean ) = {||}
var USER-NAME: map( user, string )  = {||}


rule Add-Book-To-Library( author : string,
                          title : string,
                          subject : set( string ) )
    empty( { b | (b: book) book( b ) &
                          title-of-book( b ) = title } )
 -->
        let( new-book: book = make-object( 'book ) )
           author-of-book ( new-book ) <- author;
           title-of-book  ( new-book ) <- title;
           subject-of-book( new-book ) <- subject;
```

```
            on-shelf        ( new-book ) <- true;
            book-out        ( new-book ) <- false


rule Remove-Book-From-Library( book-title : string )
   ~empty( { b | (b: book) book( b ) &
                           title-of-book( b ) = book-title } )
 -->
         let( book-to-delete: book = arb( { b | (b:book)
                 book(b) & title-of-book(b) = book-title } )
            )
            erase-object( book-to-delete )


rule Add-User( users-name : string,
               on-staff   : boolean )
   empty( { u | (u: user) user(u) &
                          user-name(u) = users-name } )
 -->
      let( new-user: user = make-object( 'user ) )
           user-name( new-user ) <- users-name;
           staff(     new-user ) <- on-staff;
           customer(  new-user ) <- ~on-staff


rule Remove-User( users-name : string )
   ~empty( { u | (u: user) user( u ) &
                           user-name( u ) = users-name } )
 -->
      let( user-to-delete: user = arb( { u | (u: user)
              user( u ) & user-name( u ) = users-name} )
         )
         erase-object( user-to-delete )


rule Check-Out-Book( whos-asking: string,
                     users-name: string,
                     which-book: string )
   ~empty( { u | (u: user)
              user( u ) & user-name( u ) = whos-asking }
         ) &
   ~empty( { u | (u: user)
              user( u ) & user-name( u ) = users-name }
         ) &
   ~empty( { b | (b: book)
              book( b ) & title-of-book( b ) = which-book }
         )
 -->
         staff( arb( { u | (u: user)
               user( u ) & user-name( u ) = whos-asking } )
              ) &
```

```
        on-shelf( arb( { b | (b: book)
              book( b ) & title-of-book( b ) = which-book } )
                )
      -->
                let( book-to-check-out: book = arb(
                        { b | (b: book) book( b ) &
                          title-of-book( b ) = which-book } )
                   )
                   on-shelf( book-to-check-out ) <- false;
                   book-out( book-to-check-out ) <- true;
                   last-checked-out-by( book-to-check-out )
                     <- users-name


rule Return-Book( whos-asking: string, which-book: string )
   ~empty( { u | (u: user)
             user( u ) & user-name( u ) = whos-asking }
         ) &
   ~empty( { b | (b: book)
             book( b ) & title-of-book( b ) = which-book }
         )
 -->
        staff( arb( { u | (u: user)
              user( u ) & user-name( u ) = whos-asking } )
             ) &
        book-out( arb( { b | (b: book)
              book( b ) & title-of-book( b ) = which-book } )
                )
      -->
                let ( book-to-be-returned: book = arb(
                       { b | (b: book) book( b ) &
                         title-of-book( b ) = which-book } )
                    )
                    on-shelf( book-to-be-returned ) <- true;
                    book-out( book-to-be-returned ) <- false


function PRINT-BOOK-SET( set-of-books: set(book) ) =
   if( empty( set-of-books ) ) then
      format( true, "This set of books is empty ~%" )
   else
      enumerate b over set-of-books do
        format( true, "AUTHOR: ~S ~30T TITLE: ~S ~%
                                ~10T SUBJECT: ~S ~%",
                    author-of-book( b ),
                    title-of-book( b ),
                    subject-of-book( b )
              )
```

```
function PRINT-USER-SET( set-of-users: set(user) ) =
   if( empty( set-of-users ) ) then
      format( true, "This set of users is empty" )
   else
      enumerate u over set-of-users do
         format( true, user-name( u ) )



function Books-On-Subject( which-subject: string ) =
   print-book-set( { b | ( b: book)
                     book( b ) &
                     {which-subject} subset subject-of-book(b) }
                 )


function Books-By-Author( which-author: string ) =
   print-book-set( { b | ( b: book )
                     book( b ) &
                     author-of-book( b ) = which-author }
                 )


function Books-Checked-Out-By-User( whos-asking : string,
                                    users-name  : string ) =
   if( ~empty( { u | (u: user)
                 user( u ) & user-name( u ) = whos-asking }
             ) &
       ~empty( { u | (u: user)
                 user( u ) & user-name( u ) = users-name }
             ) &
       ( staff( arb( { u | (u: user)
                     user( u ) & user-name( u ) = whos-asking } )
              ) or
             whos-asking = users-name ) )
   then
      print-book-set( { b | (b: book)
                        book( b ) &
                        book-out( b ) &
                        users-name = last-checked-out-by( b ) } )
   else
      format( true,
         "You are not authorized access to that information"
            )
```

Appendix H. *Visual Refine Specification for Place's Solution to Kemmer's Library Problem*

This appendix contains a complete Visual Refine Representation for the Refine Specification to Kemmers's library problem found in Appendix G.

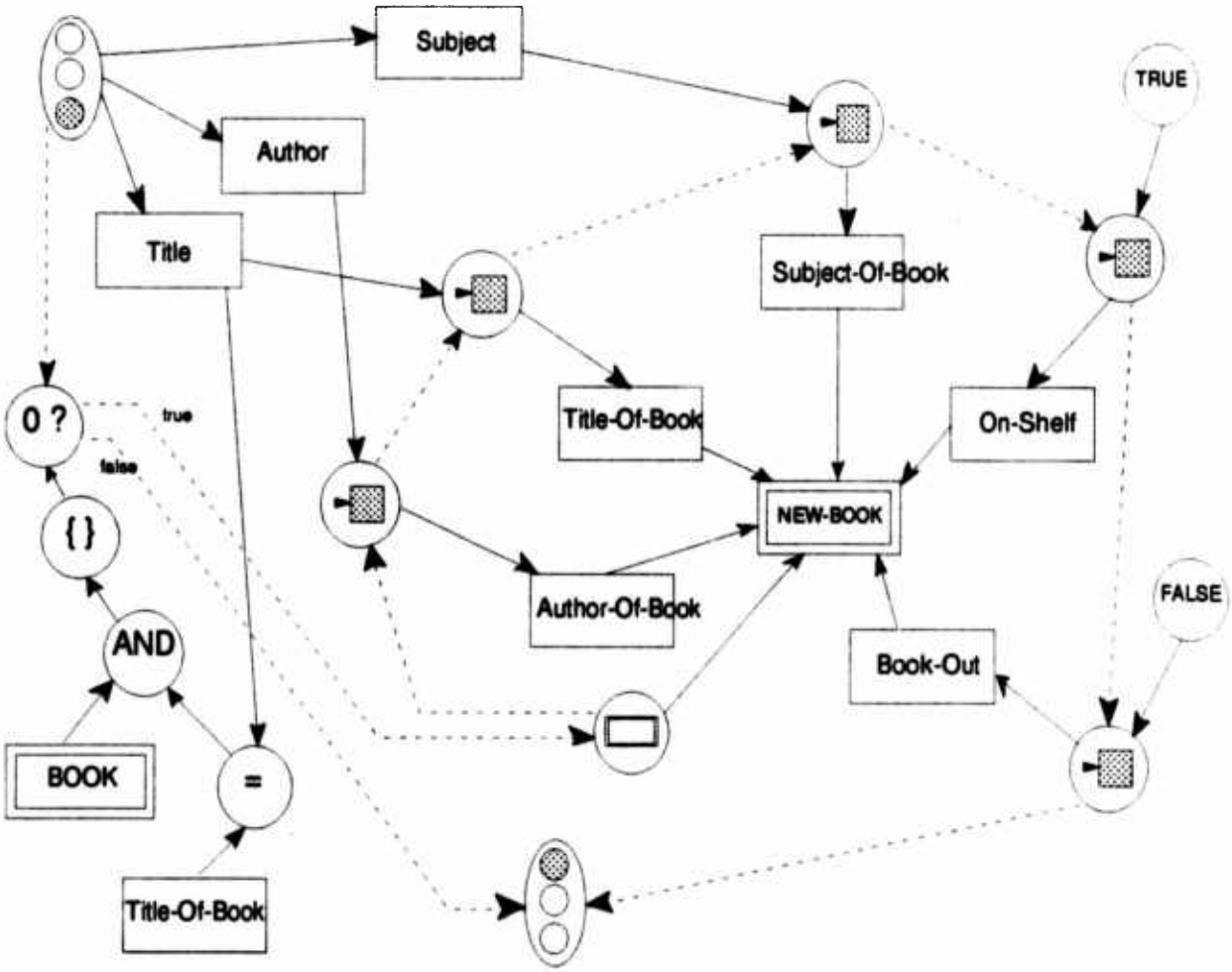


Figure H.1. Visual Refine: Add-Book-To-Library

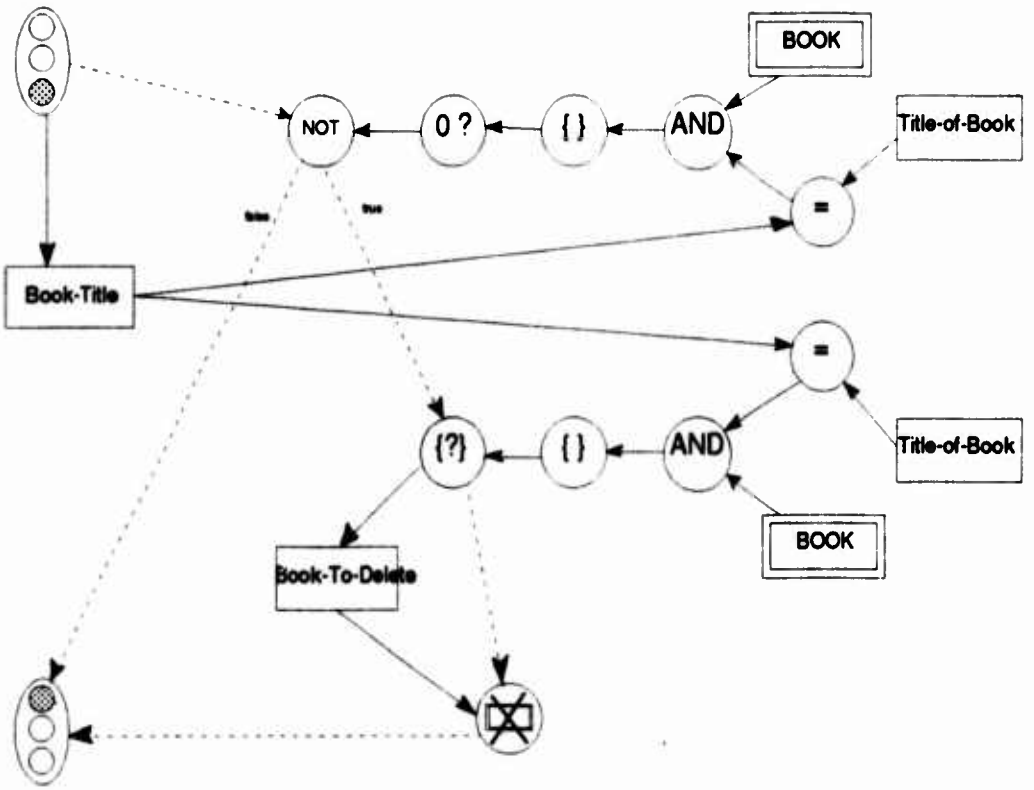


Figure H.2. Visual Refine: Remove-Book-From-Library

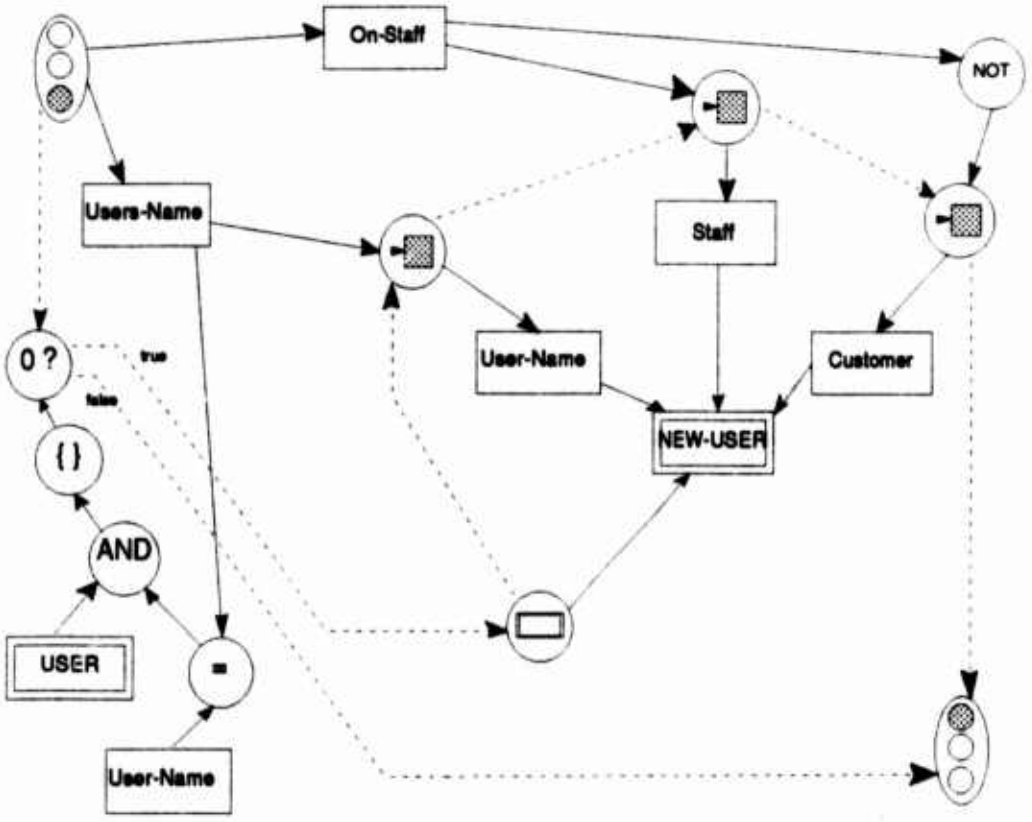


Figure H.3. Visual Refine: Add-User

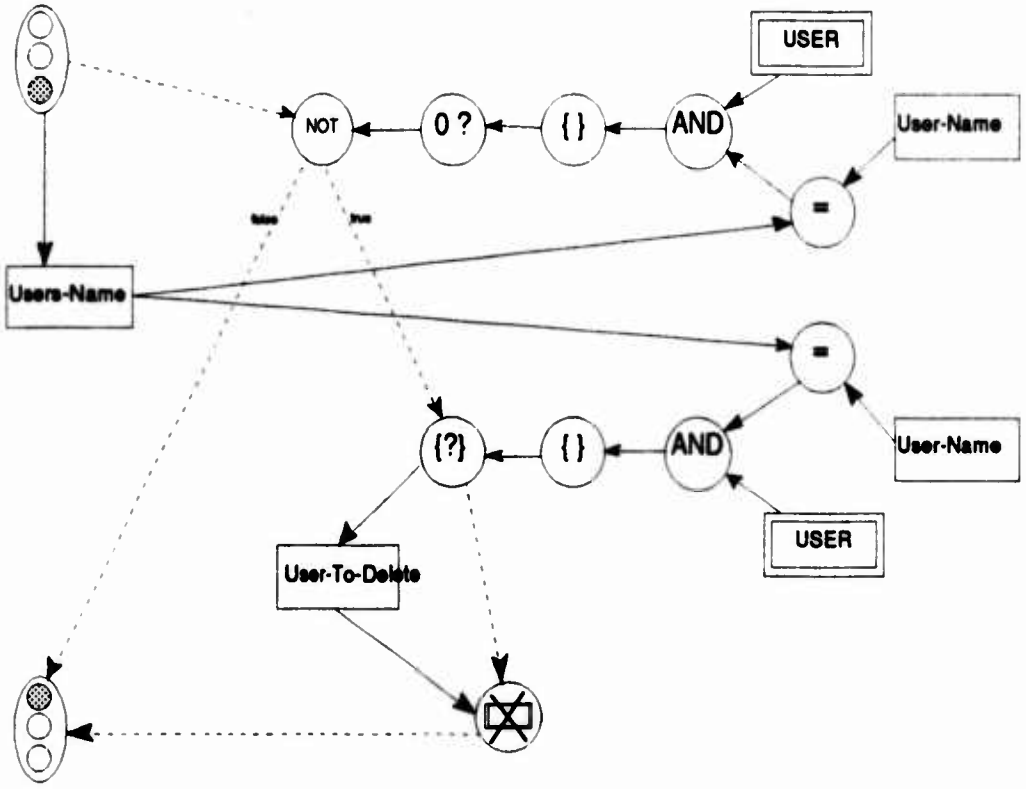


Figure H.4. Visual Refine: Remove-User

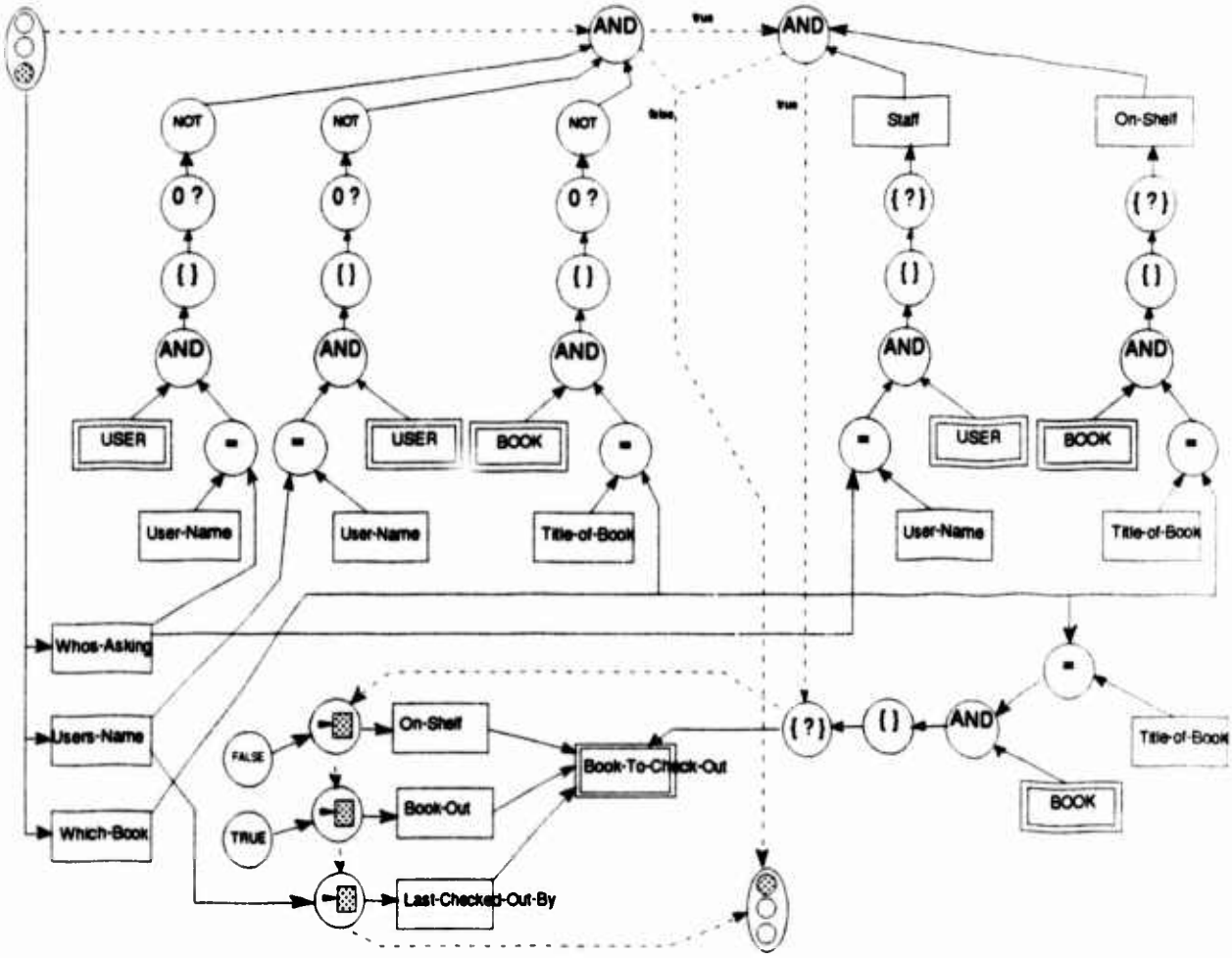


Figure H.5. Visual Refine: Check-Out-Book

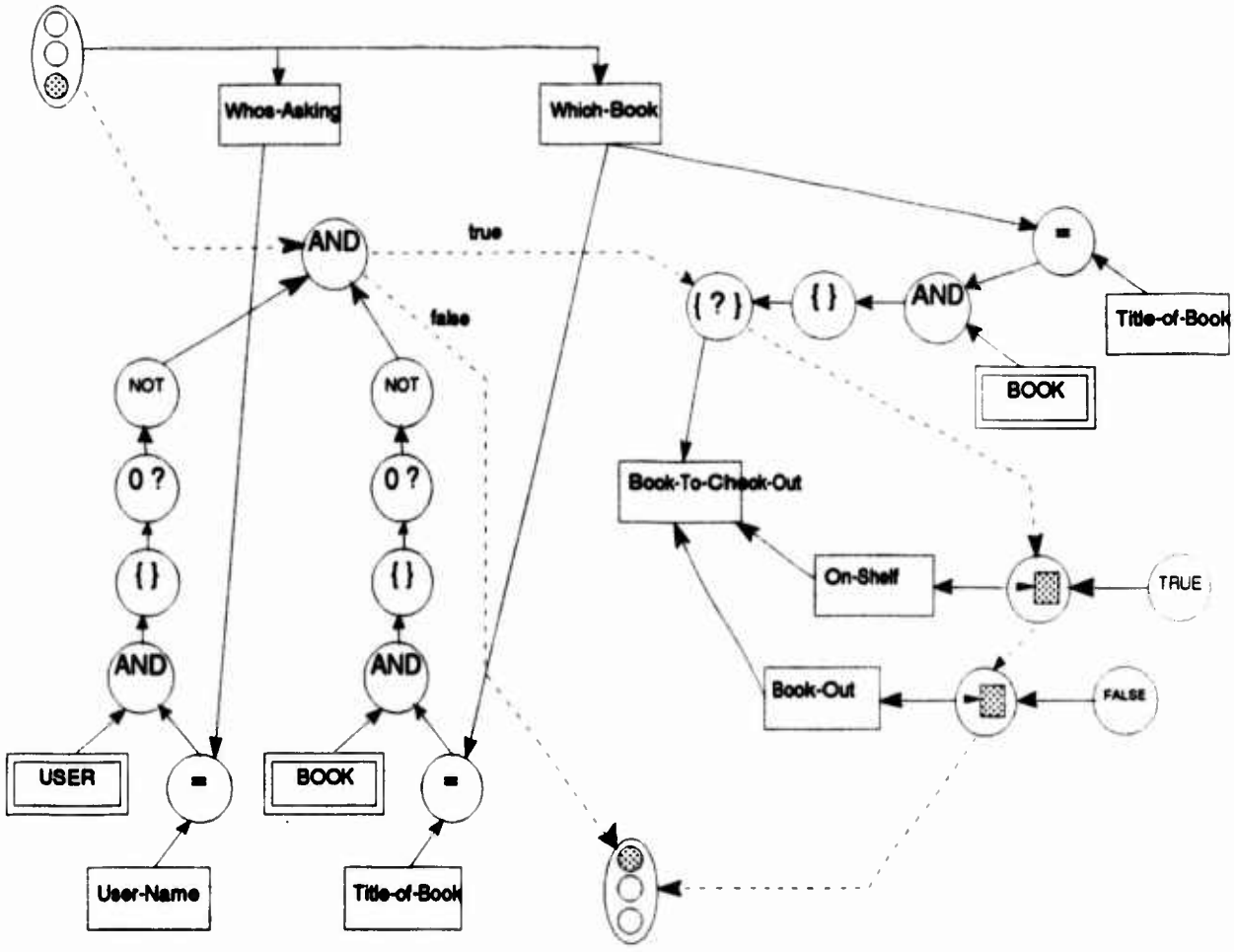


Figure H.6. Visual Refine: Return-Book

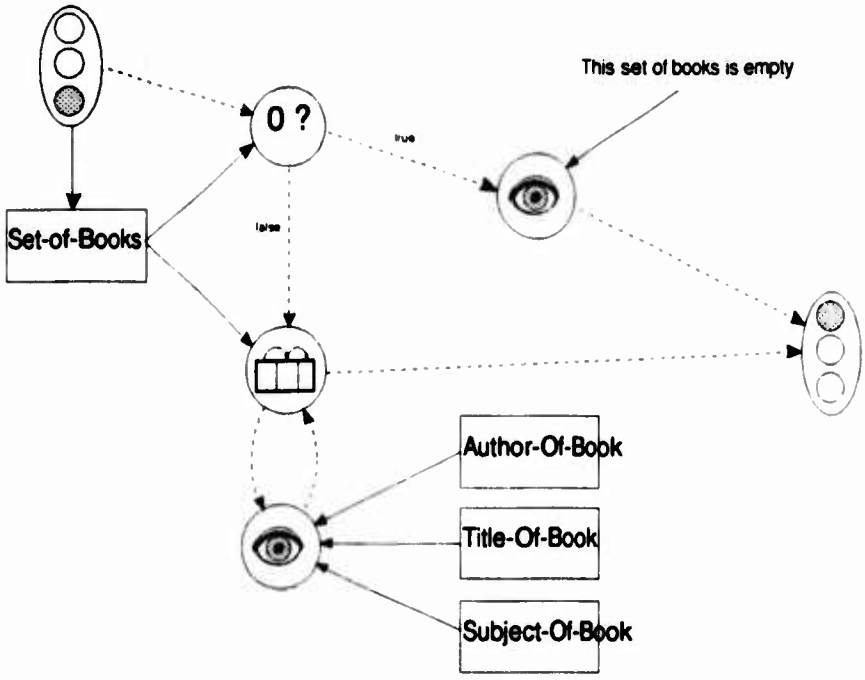


Figure H.7. Visual Refine: Print-Book-Set

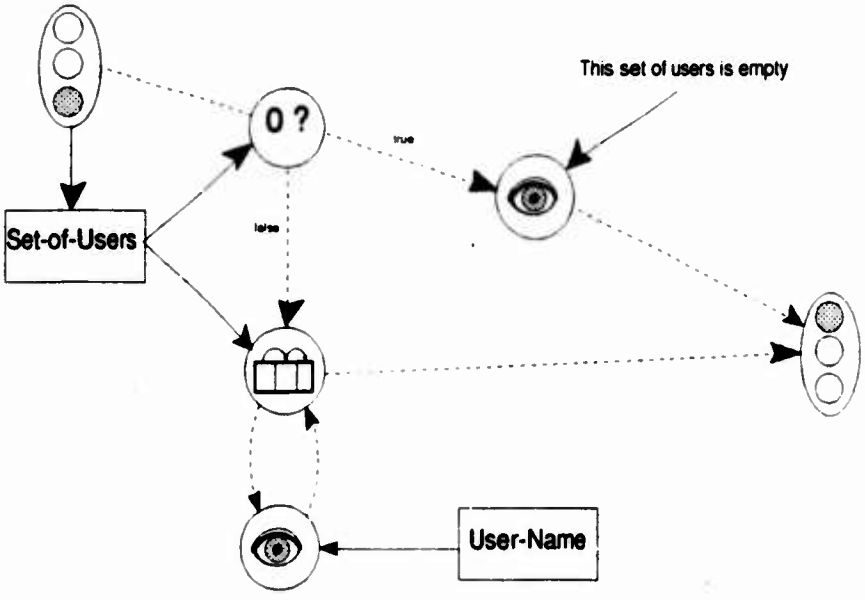


Figure H.8. Visual Refine: Print-User-Set

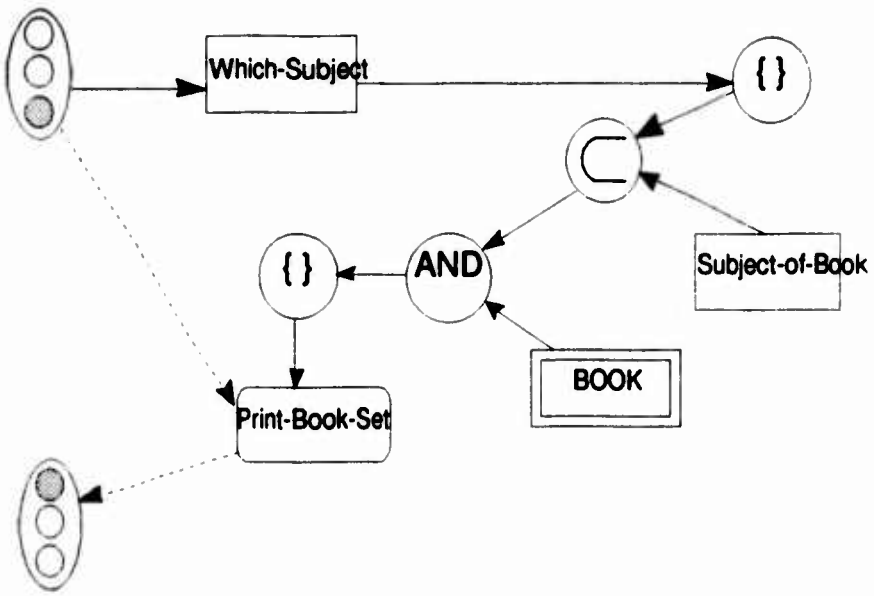


Figure H.9. Visual Refine: Books-On-Subject

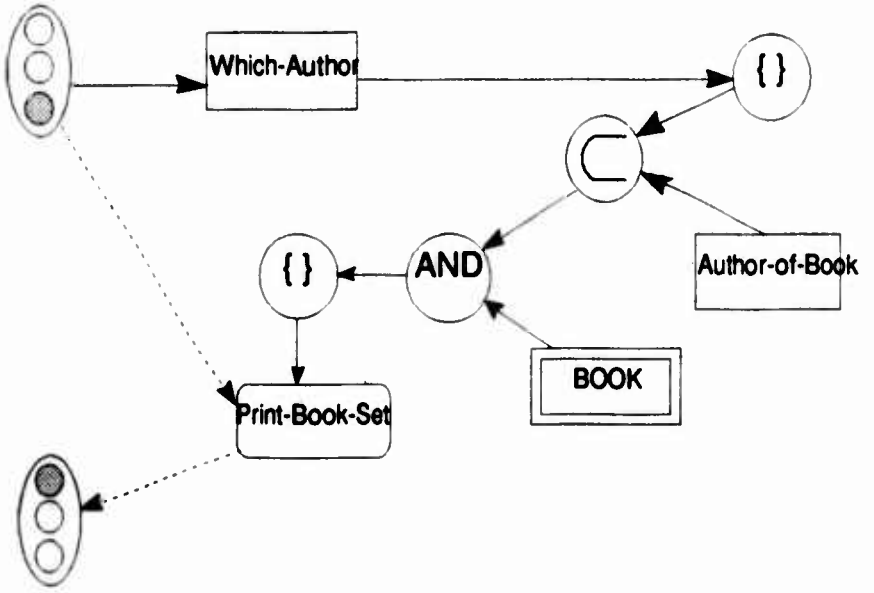


Figure H.10. Visual Refine: Books-By-Author

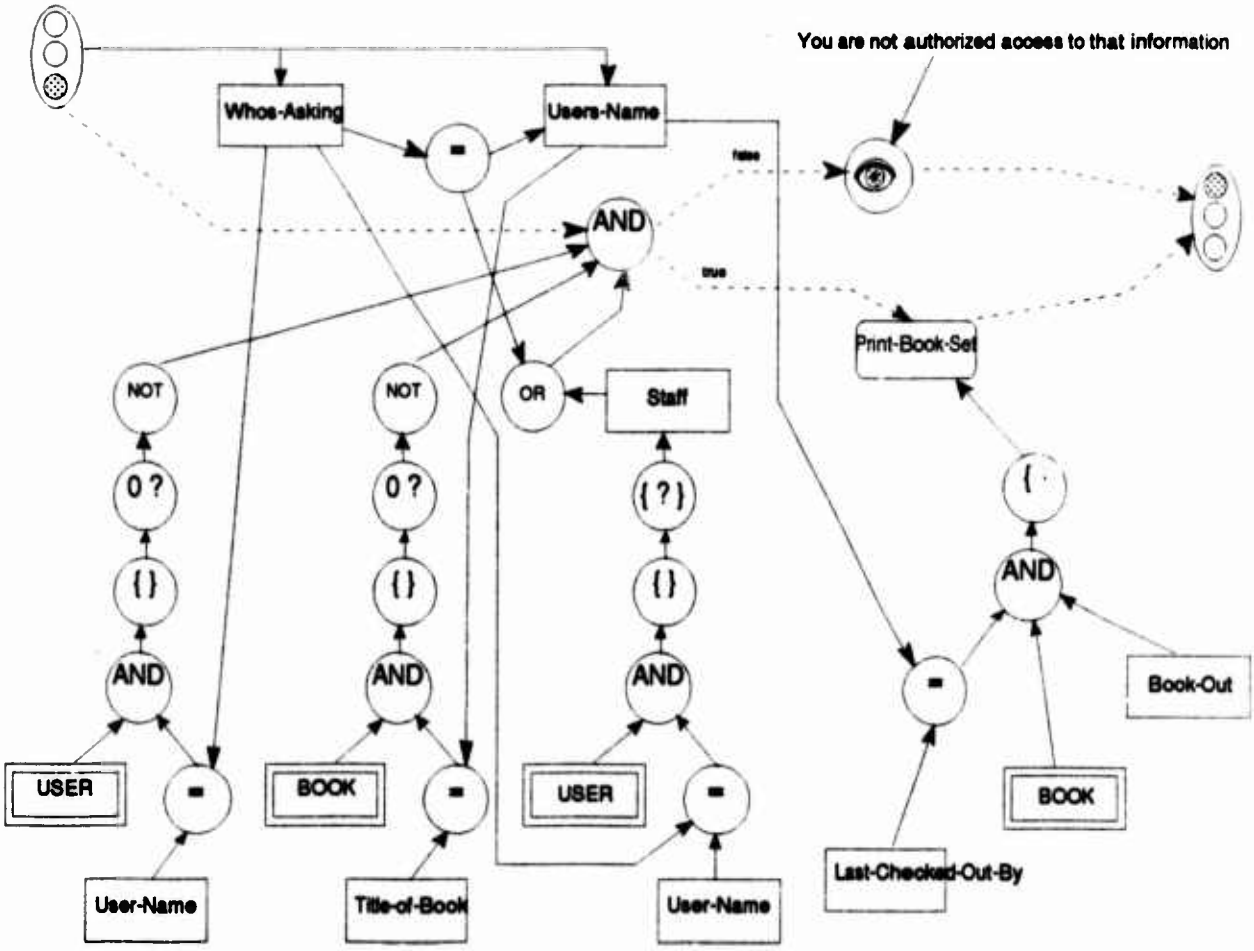


Figure H.11. Visual Refine: Books-Checked-Out-By-User

Appendix I. *Blankenship's Refine^{TM} Specification for Kemmerer's Library Problem*

This appendix contains a Refine specification for Kemmerer's library problem discussed in Chapter IV. This specification is a slightly modified version of Blankenship's Refine specification found in (6:Appendix db). The 'set-attr' Refine construct is not supported in Visual Refine and had to be modified into a block of assignment statements whenever used by Blankenship. Additionally, Blankenship used the Refine symbol data type in several places. Visualization of symbols is also not supported in Visual Refine; therefore symbols were converted to strings as necessary to allow for a visualization of Blankenship's library problem solution.

I.1 Declarations

```
!! in-package("RU")
!! in-grammar('user)

constant BOOK-LIMIT:integer = 5

% --------------
var Library-World: object-class subtype-of user-object

var Library-System: object-class subtype-of Library-World

var User:       object-class subtype-of Library-World
var Book:       object-class subtype-of Library-World

% --------------
var Name:         map( User, string )     = {||}
var Book-Count:   map(User, integer )     = {||}
var U-Type:       map( User, string )     = {||}
%% changed U-Type from symbol to string for Visual Refine
%% { 'Ordinary, 'Staff }

% --------------
var Key:          map( Book, string)      = {||}
var Author:       map( Book, string)      = {||}
var Title:        map( Book, string)      = {||}
var Subject:      map( Book, set(string)) = {||}
```

```
var Status:     map( Book, string)      = {||}
var Last-User: map( Book, string)      = {||}
%% changed Status from symbol to string for Visual Refine
%% { 'Available, 'Checked-out }

% --------------
var Current-User: map( Library-System, User) = {||}

% --------------
form Unique-Objects
   unique-names-class( 'User,       true ) &
   unique-names-class( 'Book,       true )

% --------------
var LS : Library-System     = make-object( 'Library-System )
```

I.2 Internal Utility Functions

```
function PP-Books(set-of-Book-objects: set(Book) ) =
   if empty(set-of-Book-objects)
      then format( true, "No Book Objects ~%" )
      else enumerate Bk over set-of-Book-objects do
           format( true, "Book = ~\\pp\\", Bk)

% --------------

function PP-User(set-of-User-objects: set(User) ) =
   if empty(set-of-User-objects)
      then format( true, "No User Objects ~%" )
      else enumerate Us over set-of-User-objects do
           format( true, "User = ~\\pp\\", Us)

% --------------

function PP-Library(set-of-Library-objects: set(Library-World) ) =
   if empty(set-of-Library-objects)
      then format( true, "No Library System Objects ~%" )
      else enumerate Lib over set-of-Library-objects do
           format( true, "Library System = ~\\pp\\", Lib)

% --------------

function Show-Lib() =
   true --> PP-Library( { LIB | ( LIB: Library-World )
                  Library-World(LIB) } )

% --------------
```

```
rule Init-User ( Us:     User,
                 Z-Name: string,
                 Z-Book-Count: integer,
                 Z-U-Type: string
               )
   ^Valid-User?(Z-Name)
 -->
   Name( Us )       = Z-Name       &
   Book-Count( Us ) = Z-Book-Count &
   U-Type( Us )     = Z-U-Type

% --------------

rule Init-Book ( Bo:        Book,
                 Z-Key:     string,
                 Z-Author:  string,
                 Z-Title:   string,
                 Z-Subject: set(string),
                 Z-Status:  string
               )
   ^Valid-Book?(Z-Key)
 -->
   Key( Bo )       = Z-Key     &
   Author( Bo )    = Z-Author  &
   Title( Bo )     = Z-Title   &
   Subject( Bo )   = Z-Subject &
   Status( Bo )    = Z-Status  &
   Last-User( Bo ) = " "

% --------------

function Valid-User?(Z-Name : string):boolean =
  ^empty({ u | (u: user) user(u) & Name(u) = Z-Name})

% --------------

function Find-User(Z-Name : string) : User =
   arb( {u | (u:user) user(u) & Name(u) = Z-Name})

% --------------

function Staff-Operator?(Z-User : user) :boolean =
  U-Type(Z-User) = "Staff"

% --------------

function Valid-Book?(Z-Key : string) : boolean =
   ^empty({ b | (b:book) book(b) & Key(b) = Z-Key })

% --------------
```

```
function Find-Book (Z-Key : string) : book =
   arb( { b | (b:book) book(b) & Key(b) = Z-Key })

% --------------

function Under-Book-Limit?(Z-User:string) :boolean =
   Book-Count(Find-User(Z-User)) < BOOK-LIMIT

% --------------

function Book-Available?(Z-Key :string) :boolean =
   Status(Find-Book(Z-Key)) = "Available"

% --------------
```

I.3 Internal Library Support Functions

```
function Check-Out(Z-User : string,
                   Z-Key  : string ) =
   Valid-User?(Z-User)       &
   Under-Book-Limit?(Z-User) &
   Book-Available?(Z-Key)
 -->
   Status(Find-Book(Z-Key)) = "Checked-Out" &
   Last-User(Find-Book(Z-Key)) = Z-User    &
   Book-Count(Find-User(Z-User)) = Book-Count(Find-User(Z-User)) + 1

% --------------

function Check-In-Book(Z-Key  : string ) =
   ~Book-Available?(Z-Key)
 -->
   Status(Find-Book(Z-Key)) = "Available" &
   Book-Count(Find-User(last-User(Find-Book(Z-Key)))) =
      Book-Count(Find-User(Last-User(Find-Book(Z-Key)))) - 1

% --------------

function Make-Book(Z-Key:string) =
   ~Valid-Book?(Z-Key)
 -->
   init-book(Make-Object('Book),
             Z-Key,
             read-a-string("Input the New Book's Author"),
             read-a-string("Input the New Book's Title"),
             read-a-set-of-strings(
```

```
                "Input the New Book's Subject Areas--One per line"),
              "Available"
             )

% --------------

function Delete-Book(Z-Key:string) =
   Valid-Book?(Z-Key) &
   Book-Available?(Z-Key)
 -->
   Erase-Object(Find-Book(Z-Key))

% --------------

function Books-By-Author(Z-Author:string)  =
   PP-Books( { b | (b:book) book(b) &
                    Author(b) = Z-Author } )

% --------------

function Books-By-Subject(Z-Subject:string) =
   PP-Books( { b | (b:book) book(b) &
                    {Z-Subject} subset Subject(b) } )

% --------------

function Borrowed-Books(Z-Name:string) =
   Valid-User?(Z-Name)
 -->
   PP-Books( { b | (b:book) book(b) &
              Last-User(b) = Z-Name } )

% --------------

function Last-Borrower(Z-Key:string) =
   Valid-Book?(Z-Key)
 -->
   PP-Books( { b | (b:book) book(b) &
              Key(b) = Z-Key } )

% --------------

function Make-User(Z-Name: string) =
   ~Valid-User?(Z-Name)
 -->
   init-user(Make-Object('User),Z-Name,0,"Ordinary")

% --------------

function Delete-User(Z-Name:string) =
   Valid-User?(Z-Name) &
```

```
   Book-Count(Find-User(Z-Name)) = 0
 -->
   Erase-Object(Find-User(Z-Name))
```

I.4 Externally Visable Library Rules

```
rule login(x:object)
   undefined?(Current-User(x))
 -->
   Current-User(x) = Find-User(read-a-string("Input Your Name"))

%---------------

rule logout(x:object)
   defined?(Current-User(x))
 -->
   Current-User(x) = undefined

%---------------

rule Add-Book(x:object)
   Staff-Operator?(Current-User(x))
 -->
   Make-Book(read-a-string("Input the New Book's Key") )

%---------------

rule Remove-Book(x:object)
   Staff-Operator?(Current-User(x))
 -->
   Delete-Book(read-a-string("Input the Book's Key to delete") )

% ----------------------

rule Add-User(x:object)
   Staff-Operator?(Current-User(x))
 -->
   Make-User(read-a-string("Input the New User's Name"))

%---------------

rule Remove-User(x:object)
   Staff-Operator?(Current-User(x))
 -->
   Delete-User(read-a-string("Input the User's Name to delete") )
```

```
% --------------

rule Check-Out-Book(x:object)
   Staff-Operator?(Current-User(x))
 -->
   Check-Out(read-a-string("Input the Person's Name"),
             read-a-string("Input the Book's Key") )

%---------------

rule Return-Book(x:object)
   Staff-Operator?(Current-User(x))
 -->
   Check-In-Book(read-a-string("Input the Book's Key to Delete"))

%---------------

rule List-Books-By-Author(x:object)
   Valid-User?(Name(Current-User(x)))
 -->
   Books-By-Author(read-a-string("Input the Desired Author's Name"))

%---------------

rule List-Books-By-Subject(x:object)
   Valid-User?(Name(Current-User(x)))
 -->
   Books-By-Subject(read-a-string("Input the Desired Subject Area"))

%---------------

rule List-Borrowed-Books(x:object)
   Staff-Operator?(Current-User(x))
 -->
   Borrowed-Books(read-a-string("Input the Desired User's Name"))

%---------------

rule List-My-Borrowed-Books(x:object)
   Valid-User?(Name(Current-User(X)))
 -->
   Borrowed-Books(Name(Current-User(X)))

%---------------

rule List-Last-Borrower(x:object)
   Staff-Operator?(Current-User(x))
 -->
   Last-Borrower(read-a-string("Input the Desired Book's Key"))
```

Appendix J. *Visual Refine Specification for Blankenship's Solution to Kemmer's Library Problem*

This appendix contains a complete Visual Refine Representation for the Refine Specification to Kemmers's library problem found in Appendix I.

J.1 Internal Utility Functions

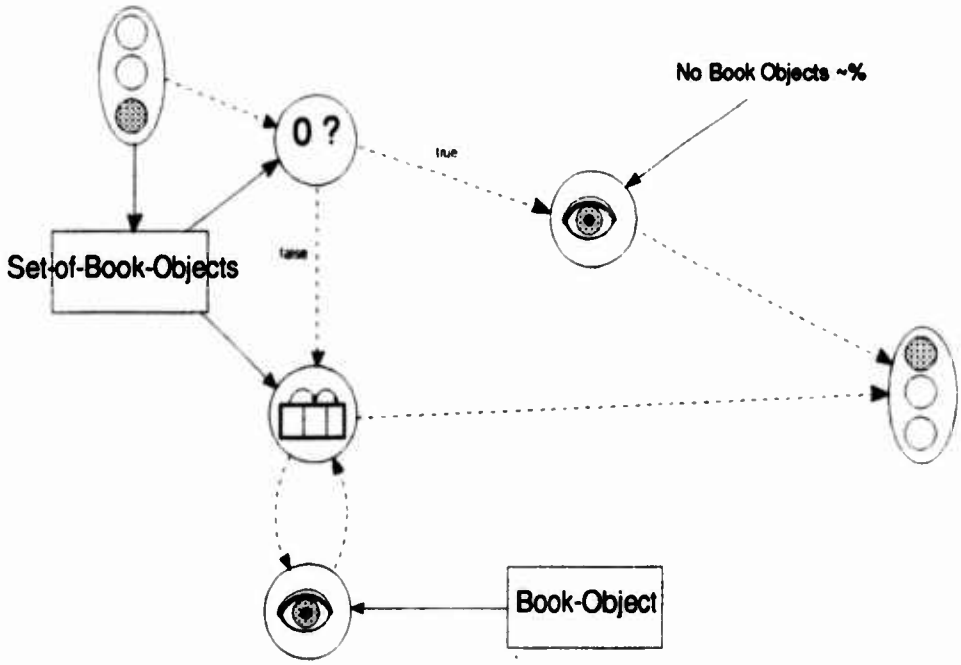


Figure J.1. Visual Refine: PP-Books

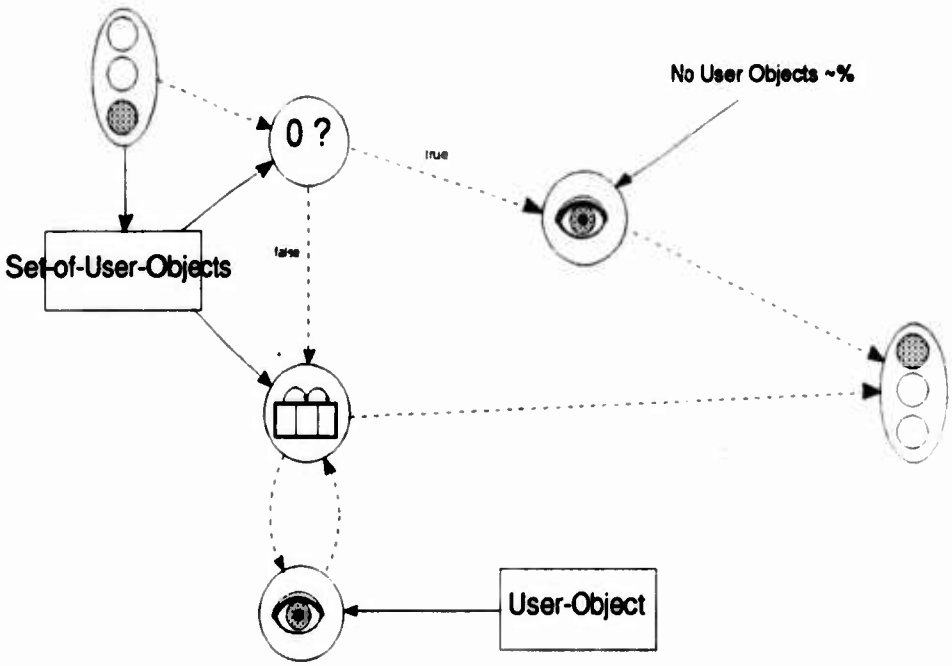


Figure J.2. Visual Refine: PP-User

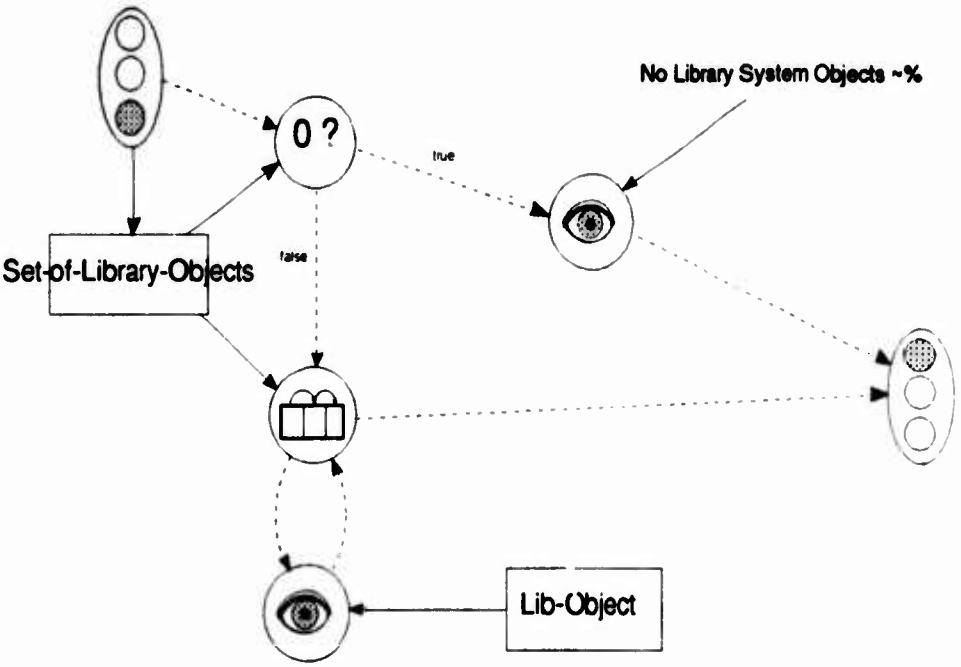


Figure J.3. Visual Refine: PP-Library

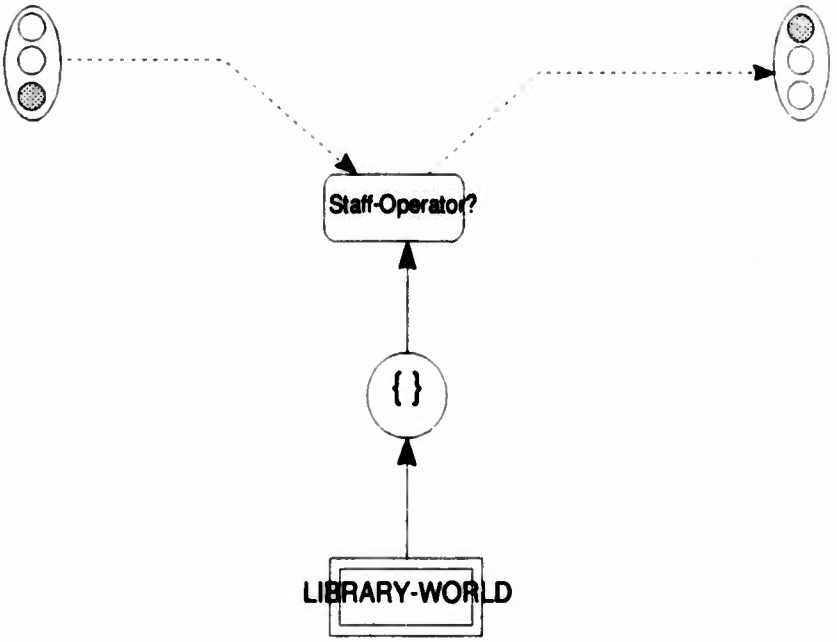


Figure J.4. Visual Refine: Show-Lib

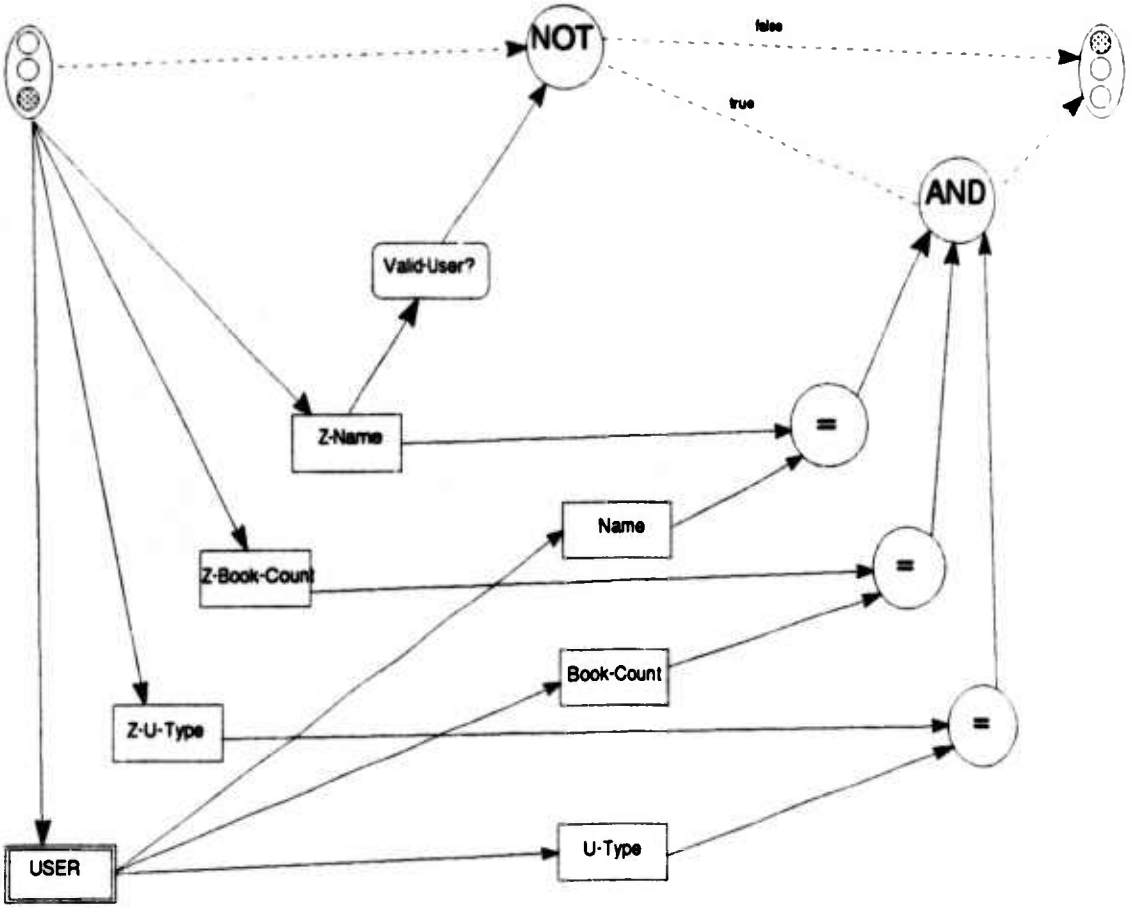


Figure J.5. Visual Refine: Init-User

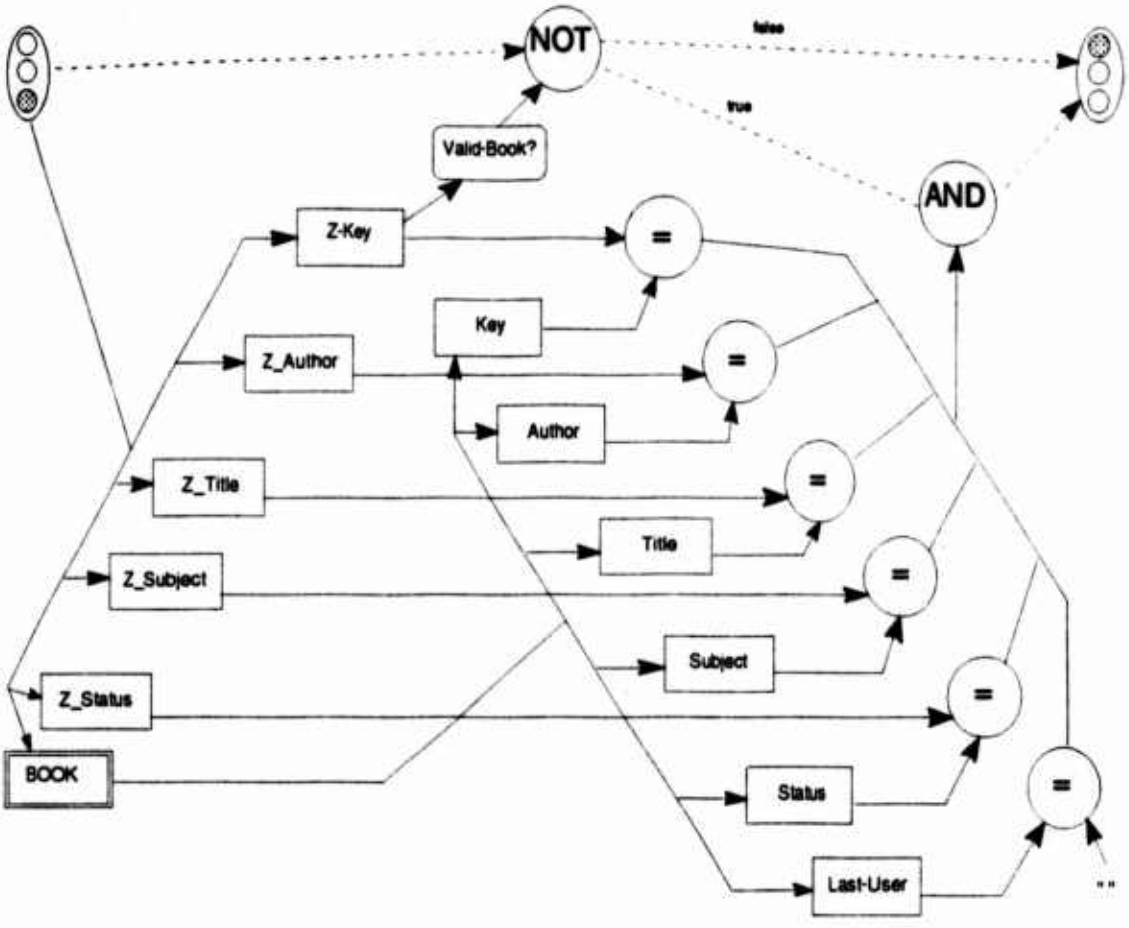


Figure J.6. Visual Refine: Init-Book

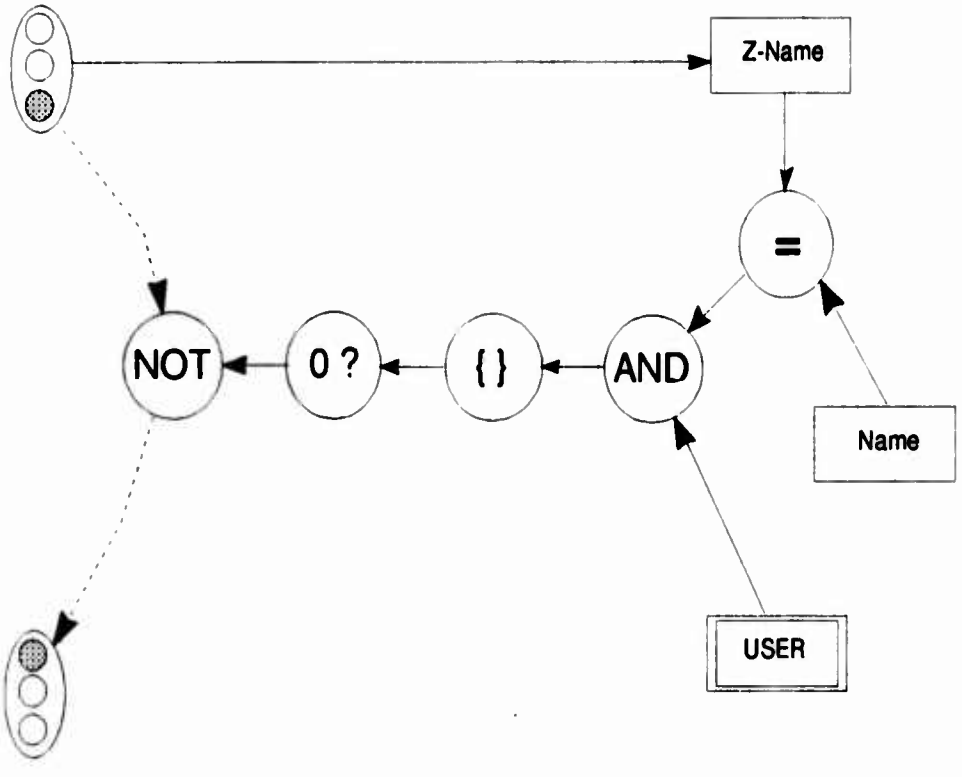


Figure J.7. Visual Refine: Valid-User?

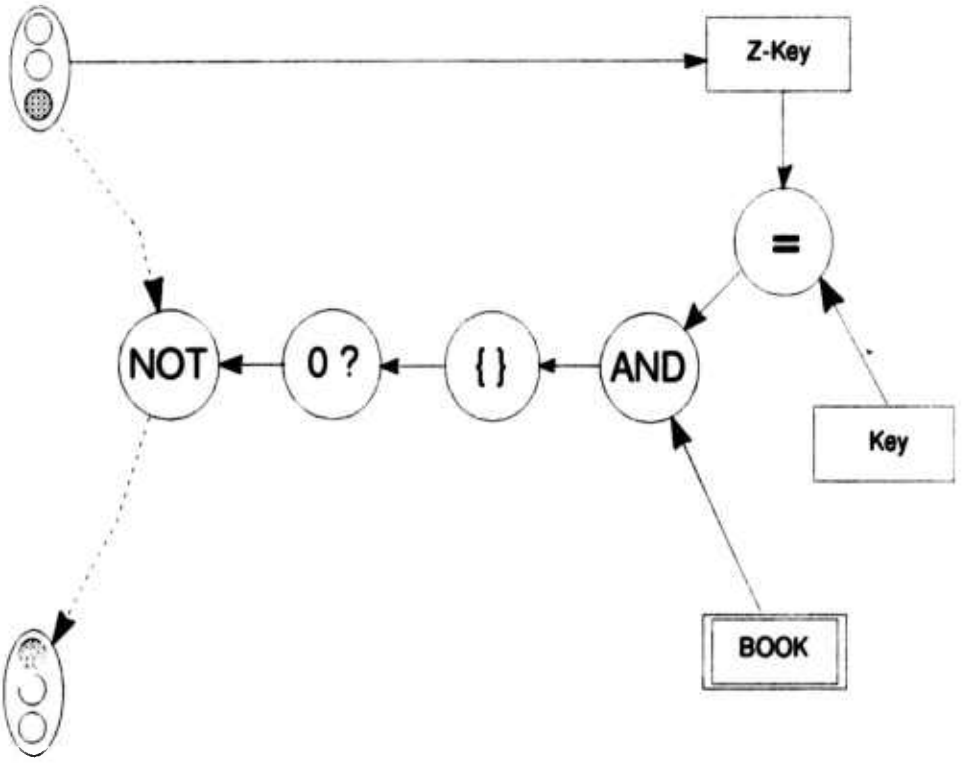


Figure J.8. Visual Refine: Valid-Book?

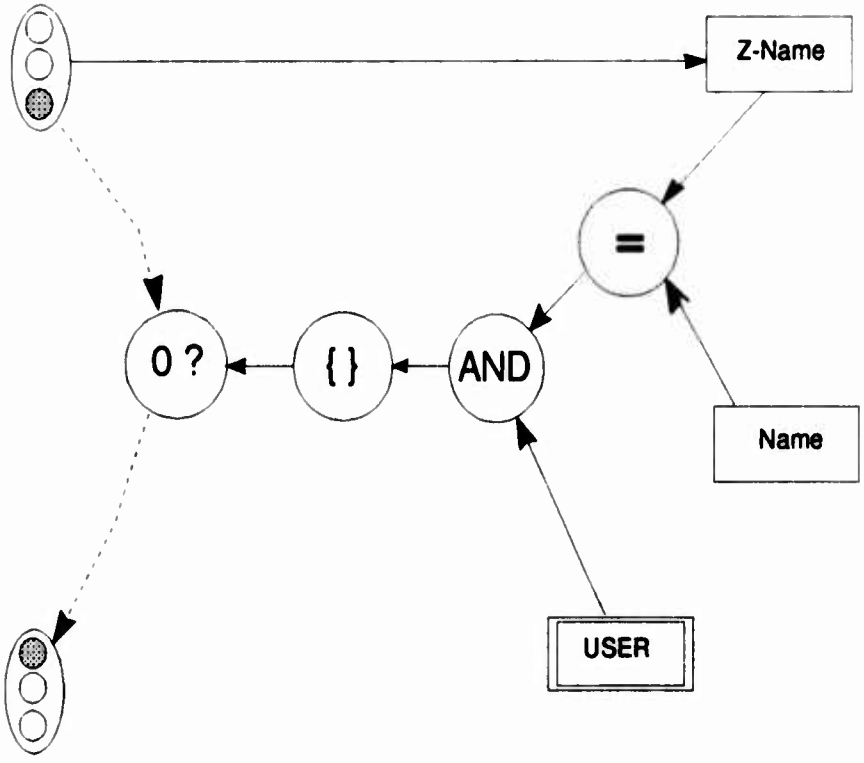


Figure J.9. Visual Refine: Find-User?

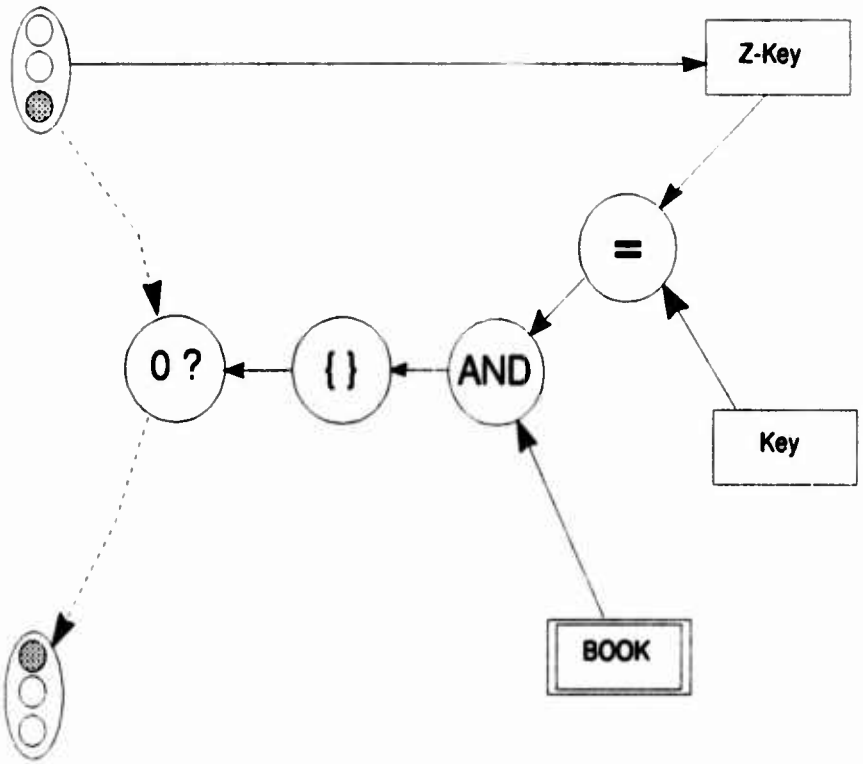


Figure J.10. Visual Refine: Find-Book?

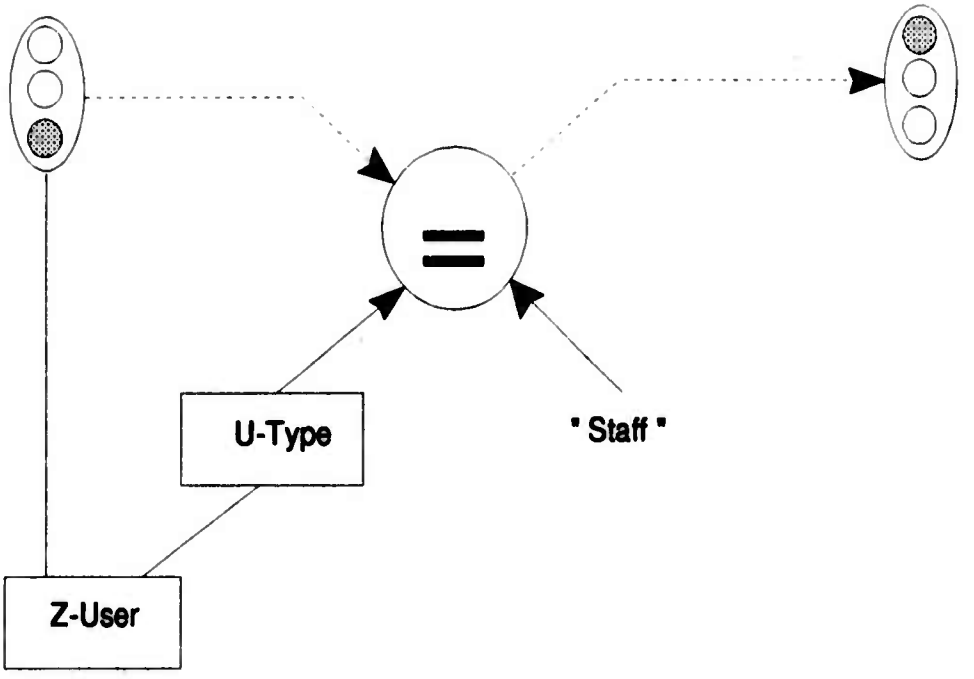


Figure J.11. Visual Refine: Staff-Operator?

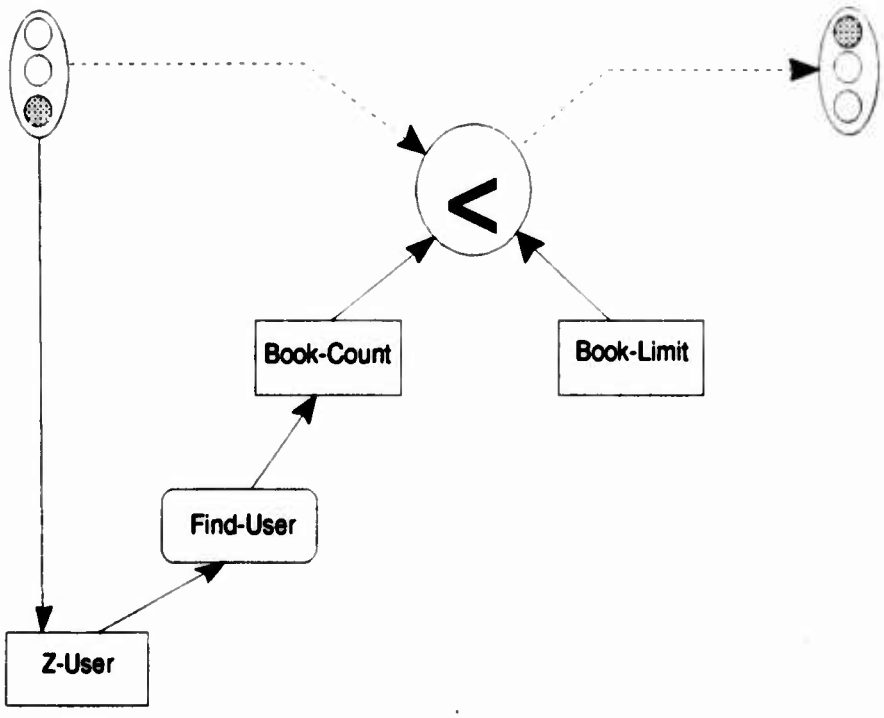


Figure J.12. Visual Refine: Under-Book-Limit?

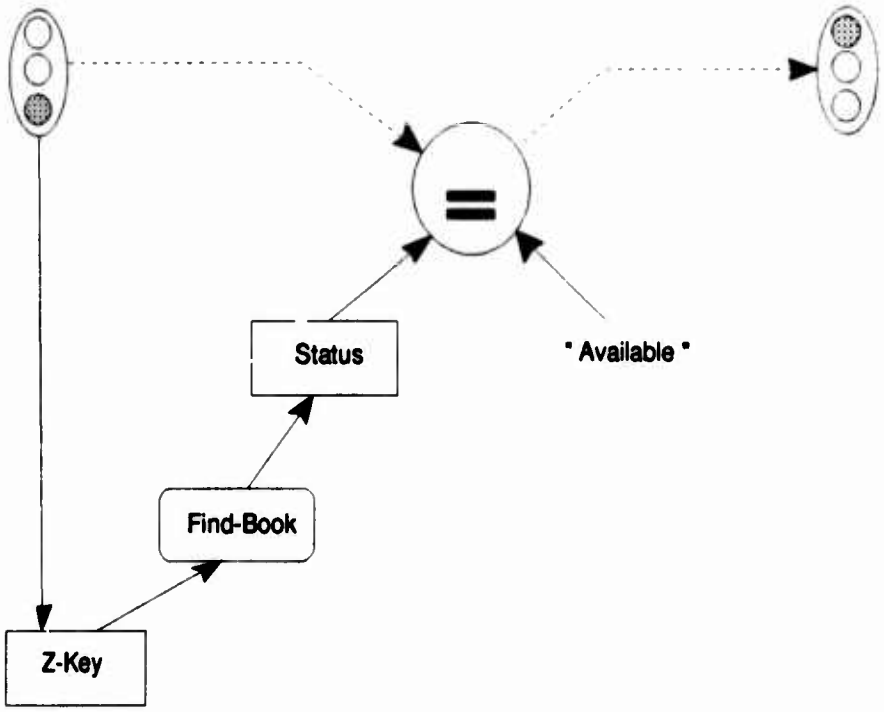


Figure J.13. Visual Refine: Book-Available?

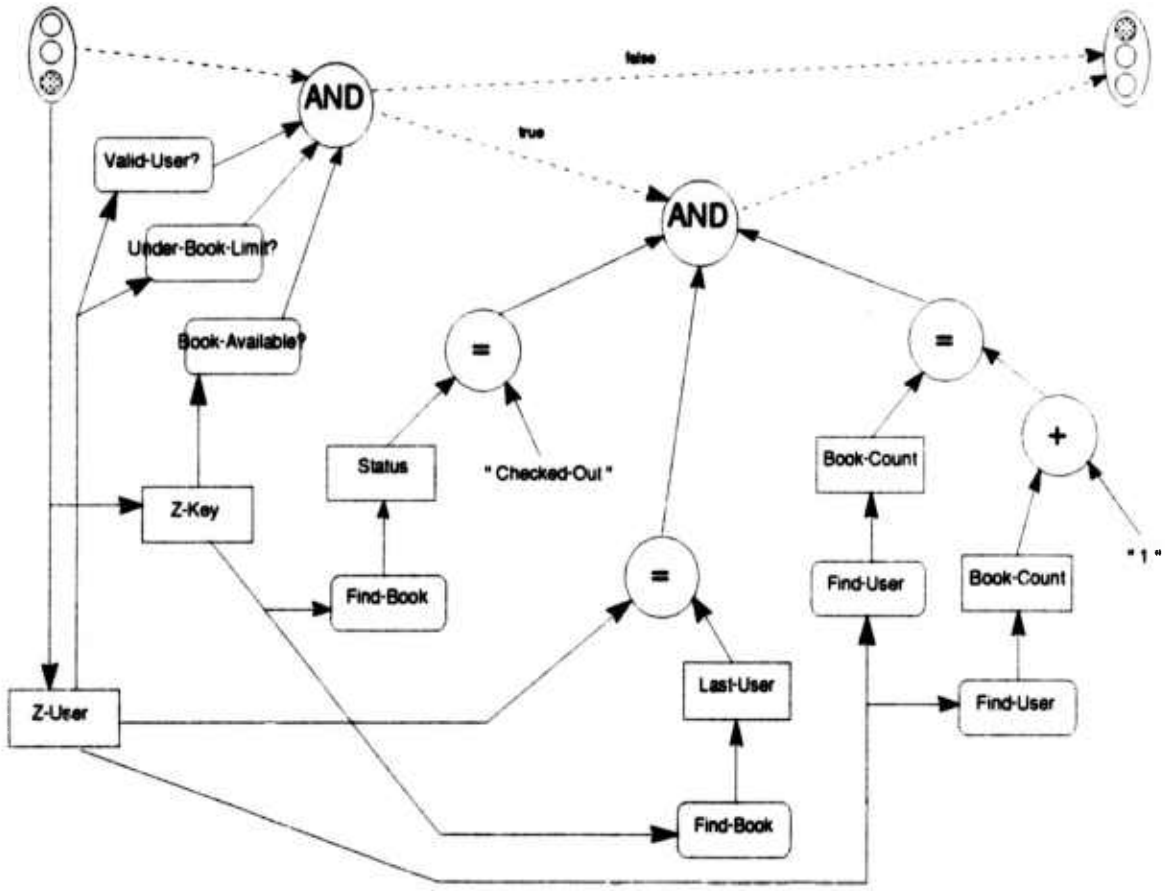


Figure J.14. Visual Refine: Check-Out

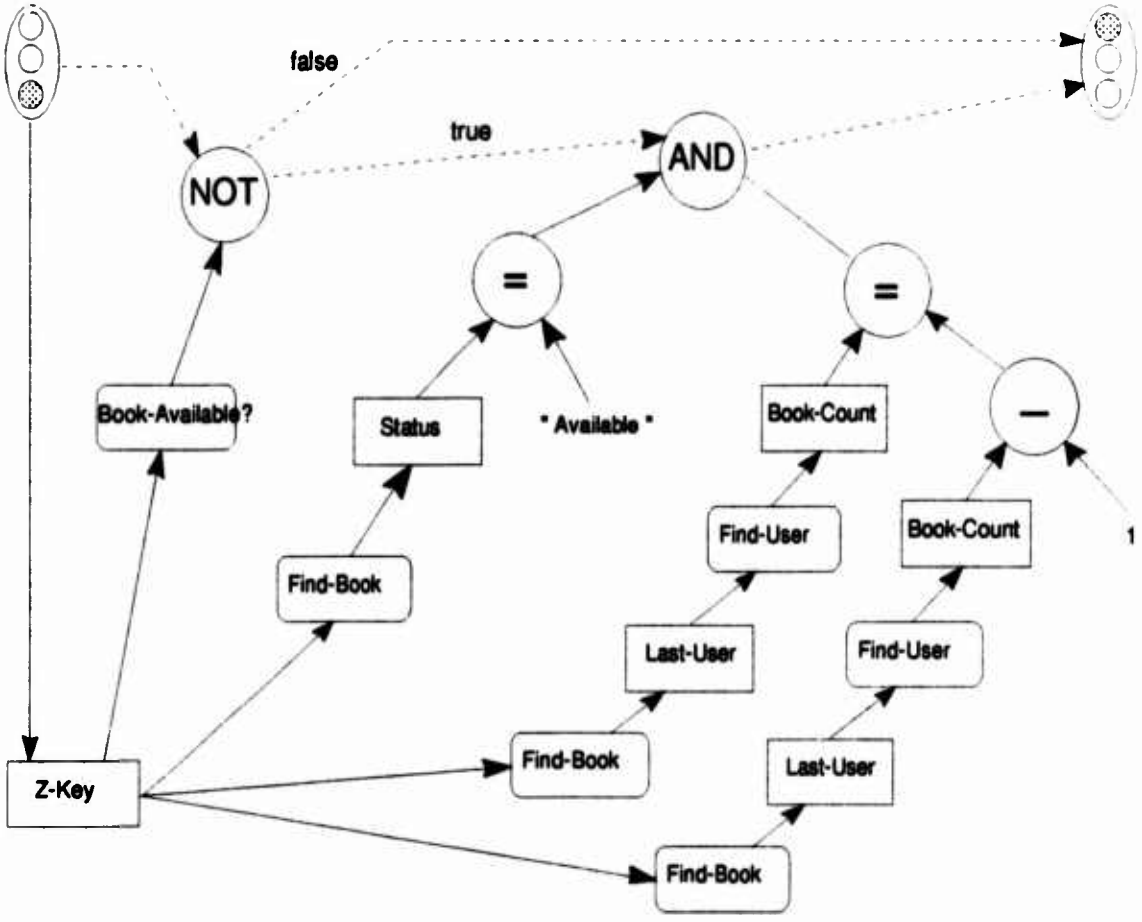


Figure J.15. Visual Refine: Check-In-Book

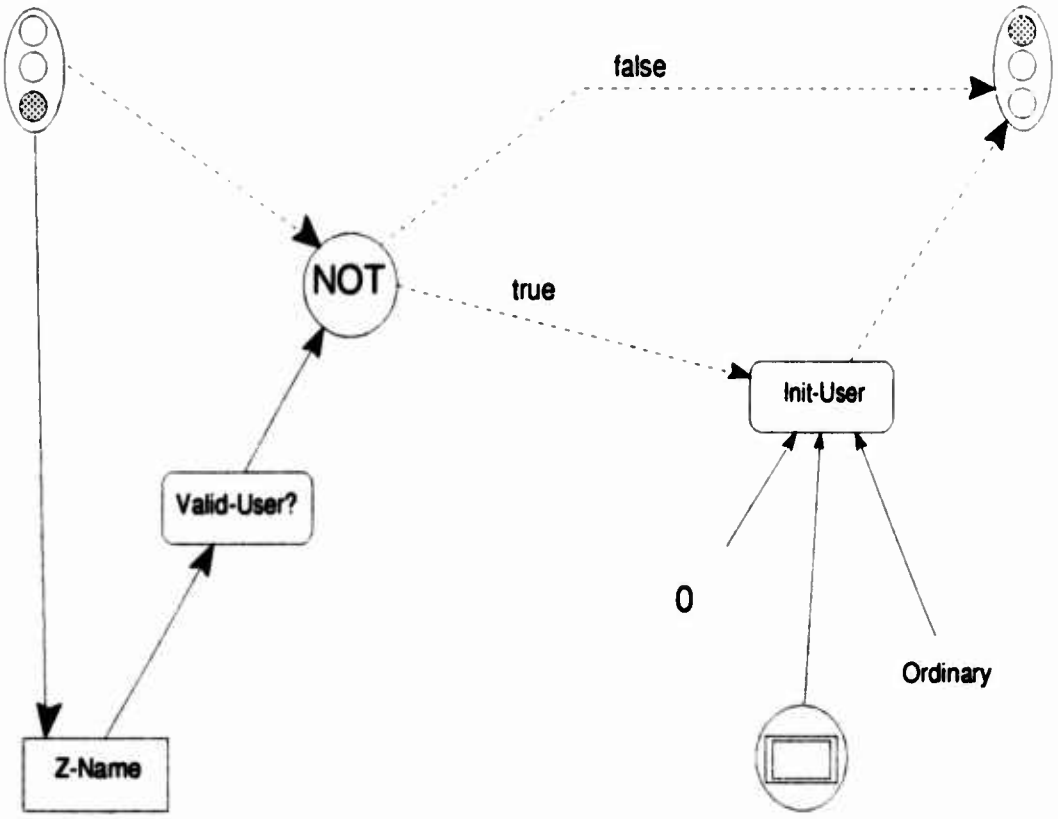


Figure J.16. Visual Refine: Make-User

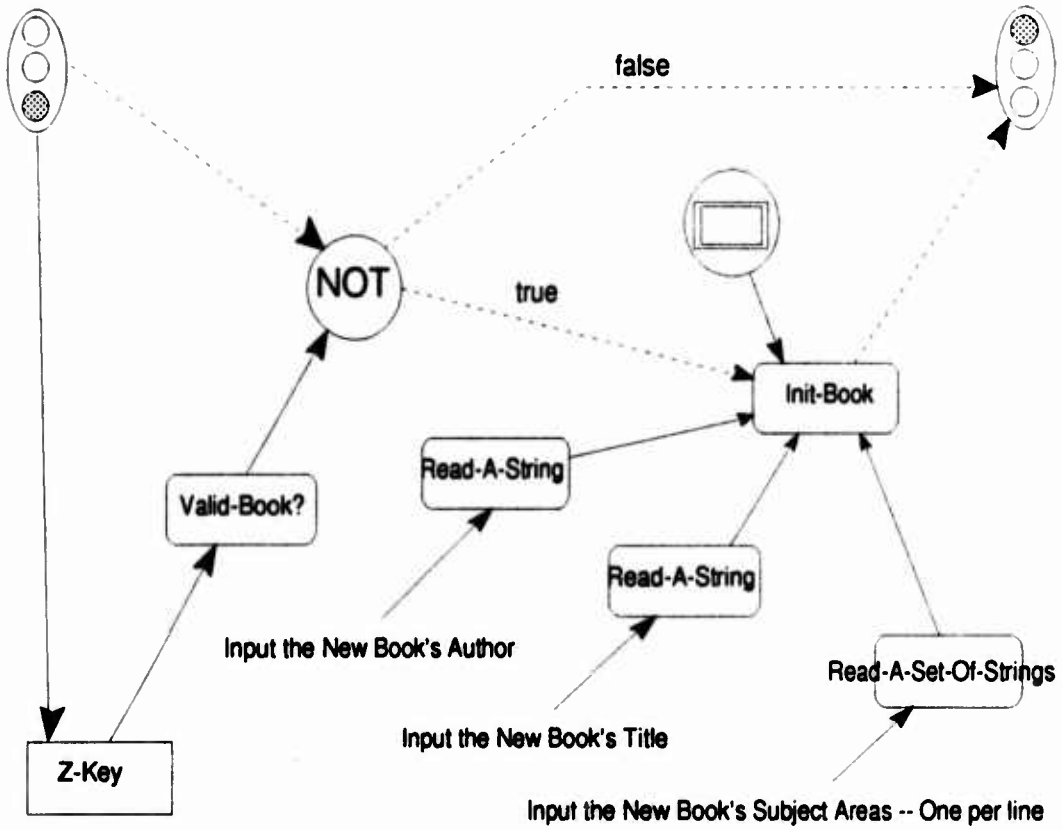


Figure J.17. Visual Refine: Make-Book

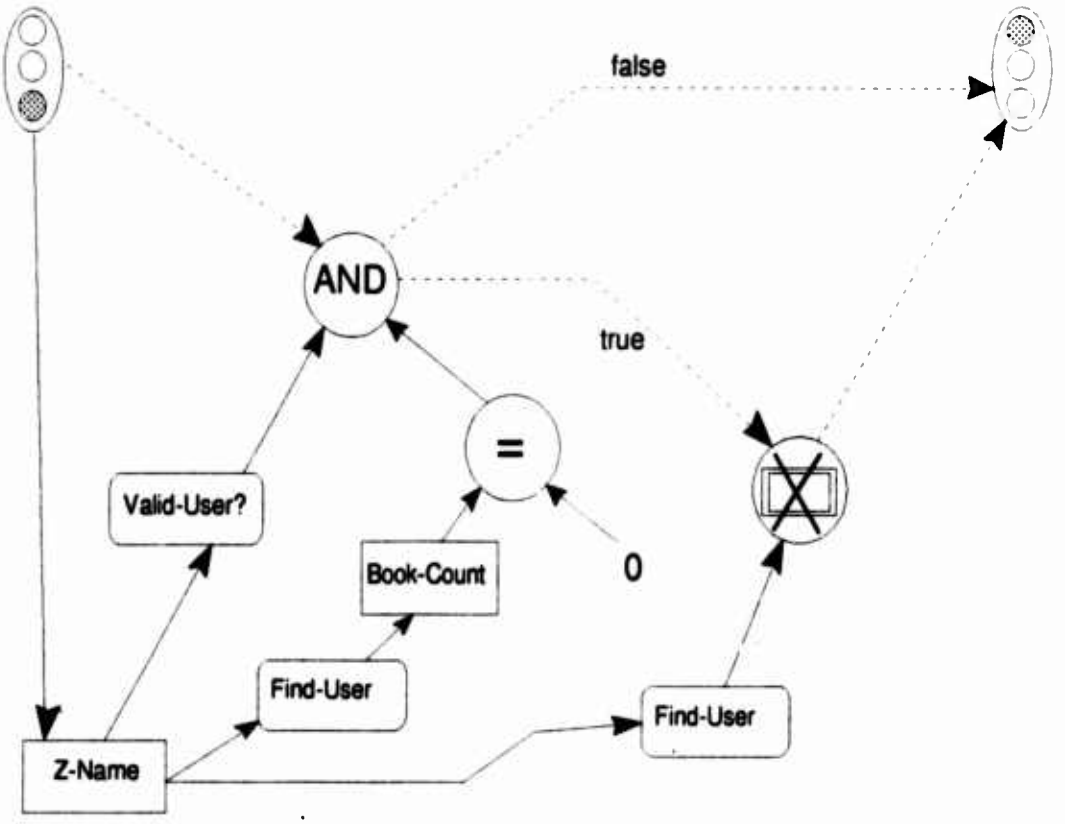


Figure J.18. Visual Refine: Delete-User

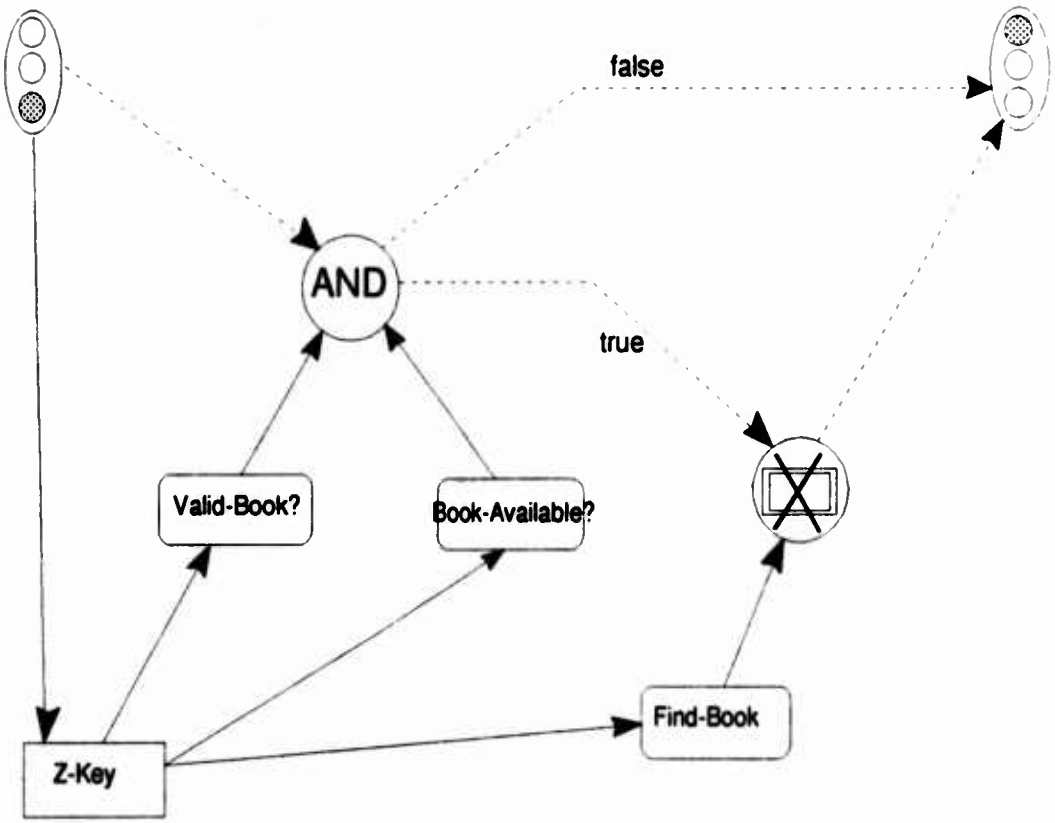


Figure J.19. Visual Refine: Delete-Book

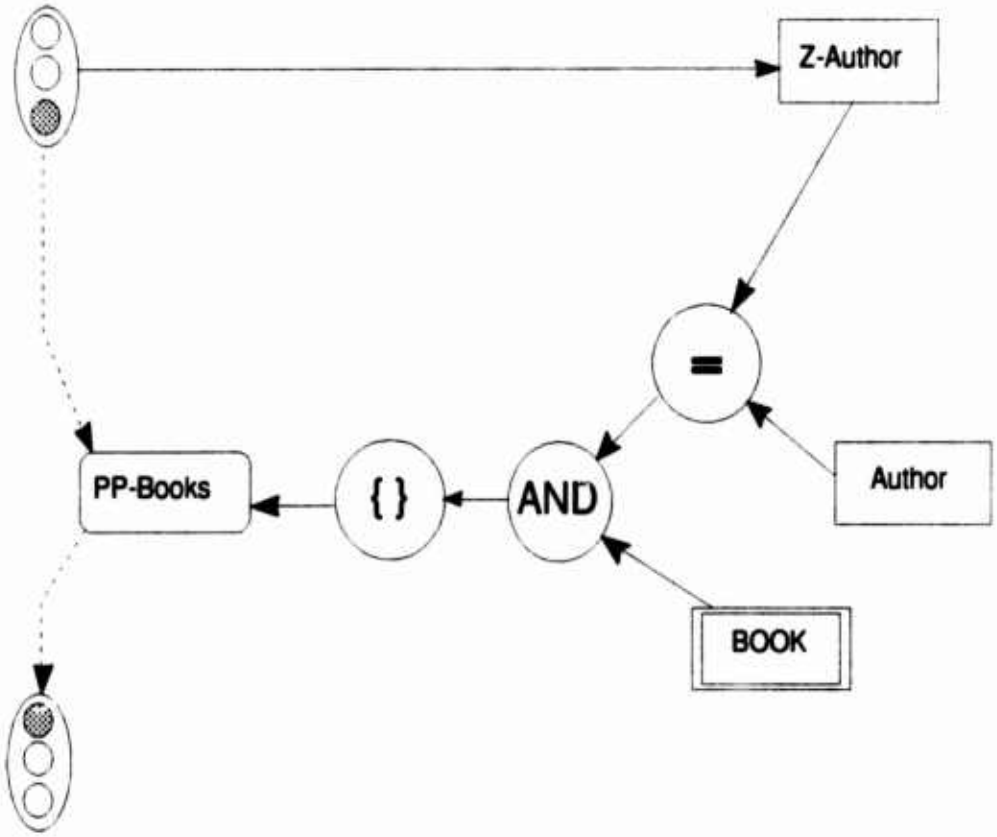


Figure J.20. Visual Refine: Books-By-Author

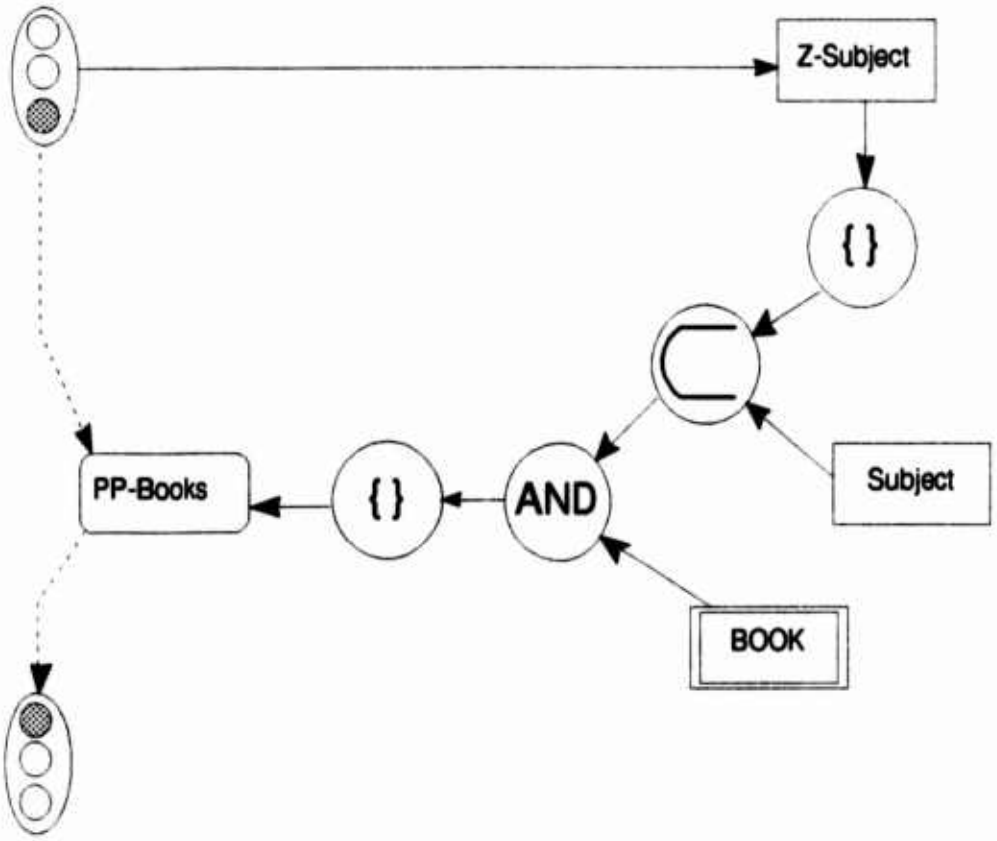


Figure J.21. Visual Refine: Books-By-Subject

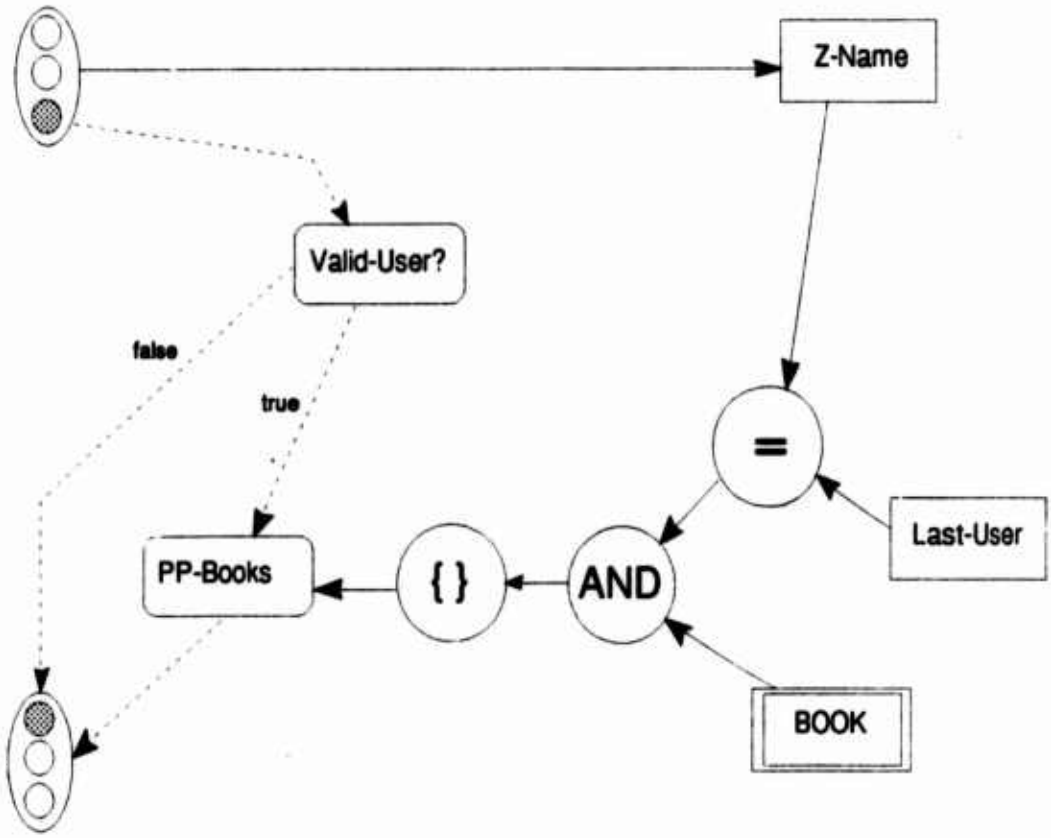


Figure J.22. Visual Refine: Borrowed-Books

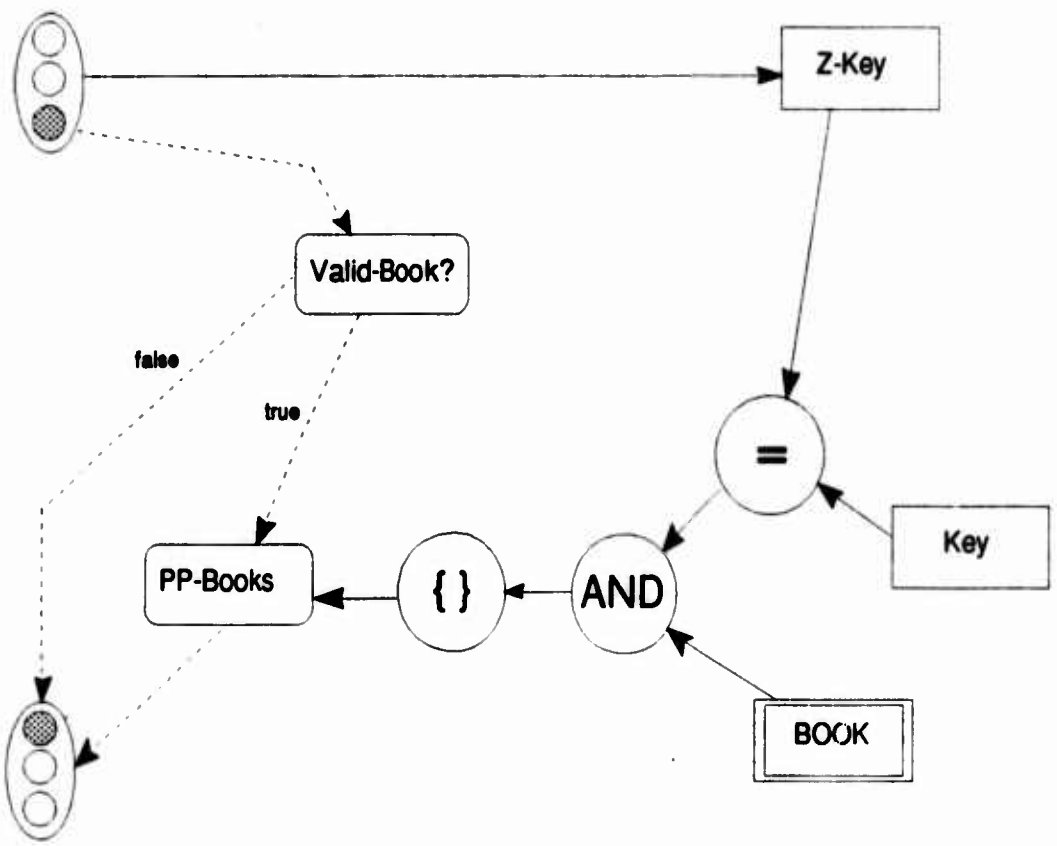


Figure J.23. Visual Refine: Last-Borrower

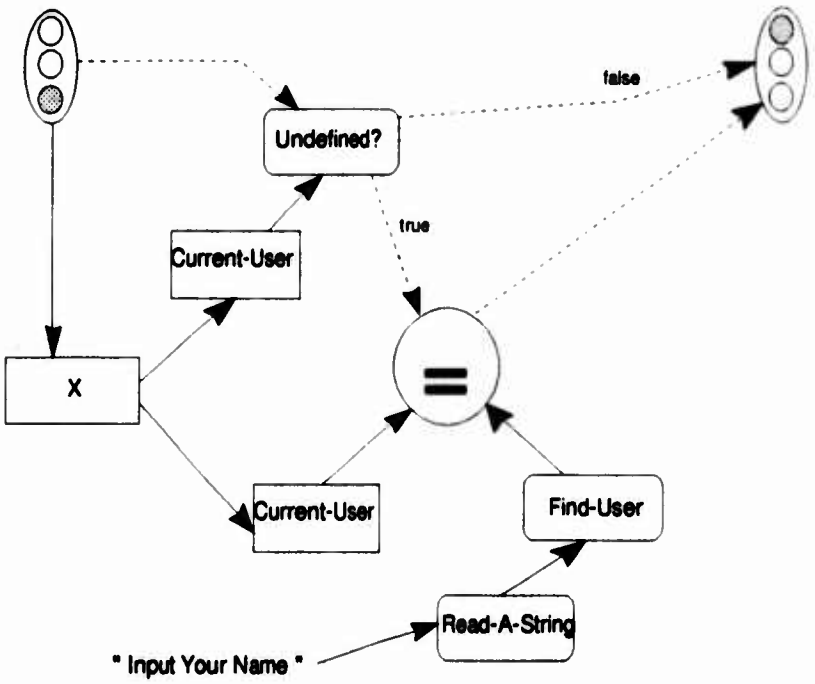


Figure J.24. Visual Refine: Login

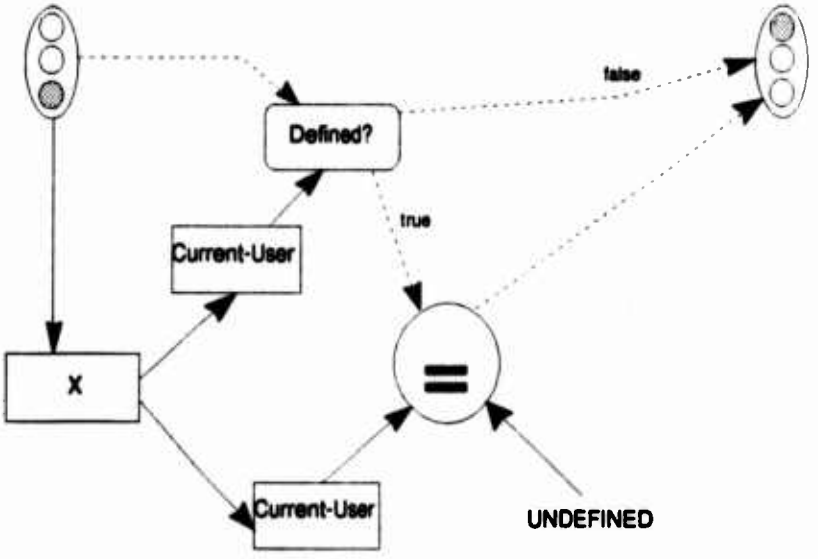


Figure J.25. Visual Refine: Logout

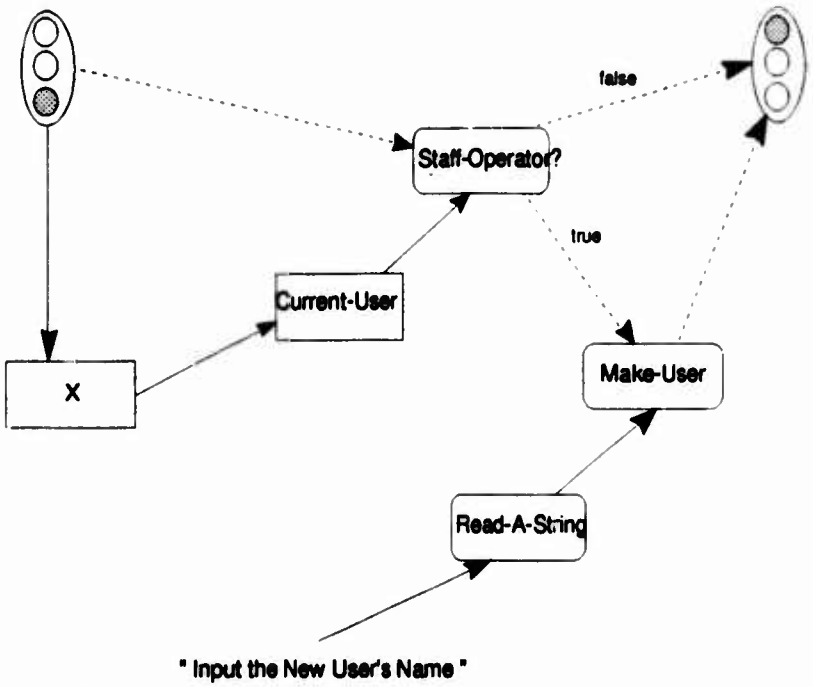


Figure J.26. Visual Refine: Add-User

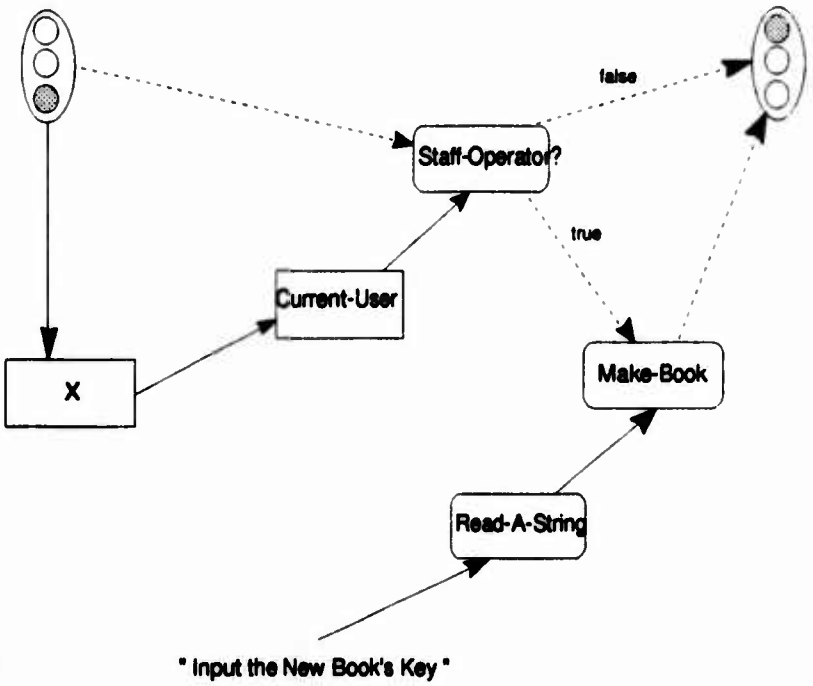


Figure J.27. Visual Refine: Add-Book

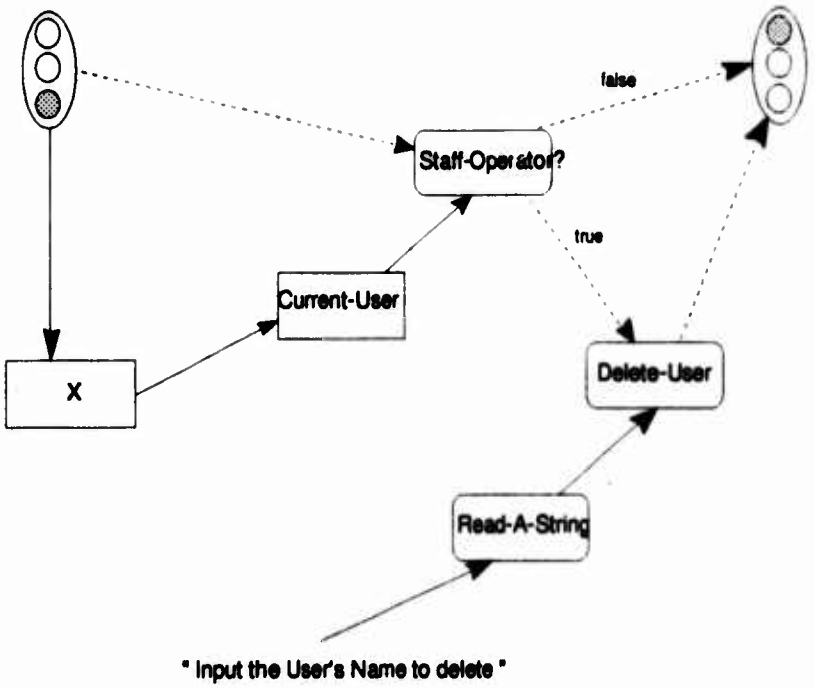


Figure J.28. Visual Refine: Remove-User

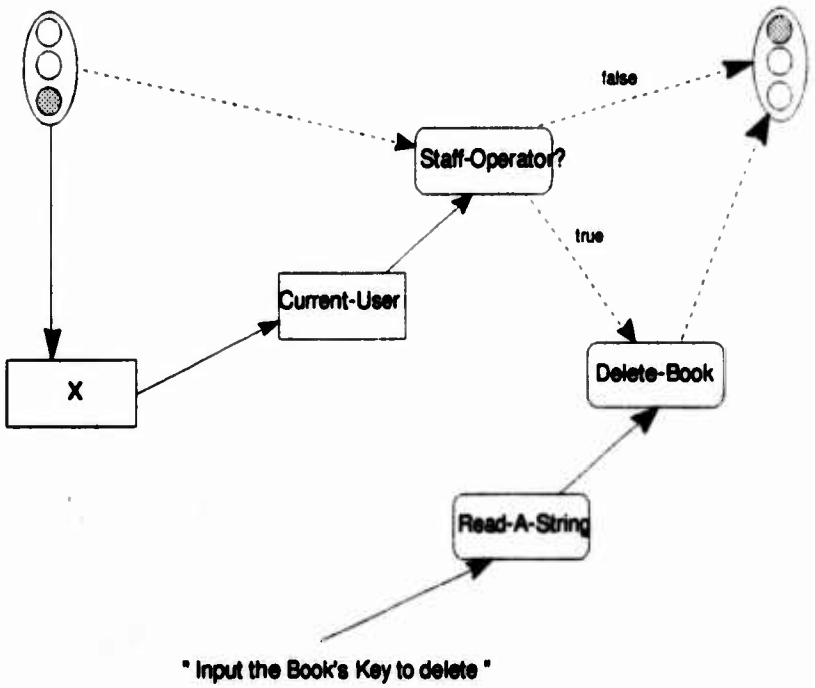


Figure J.29. Visual Refine: Remove-Book

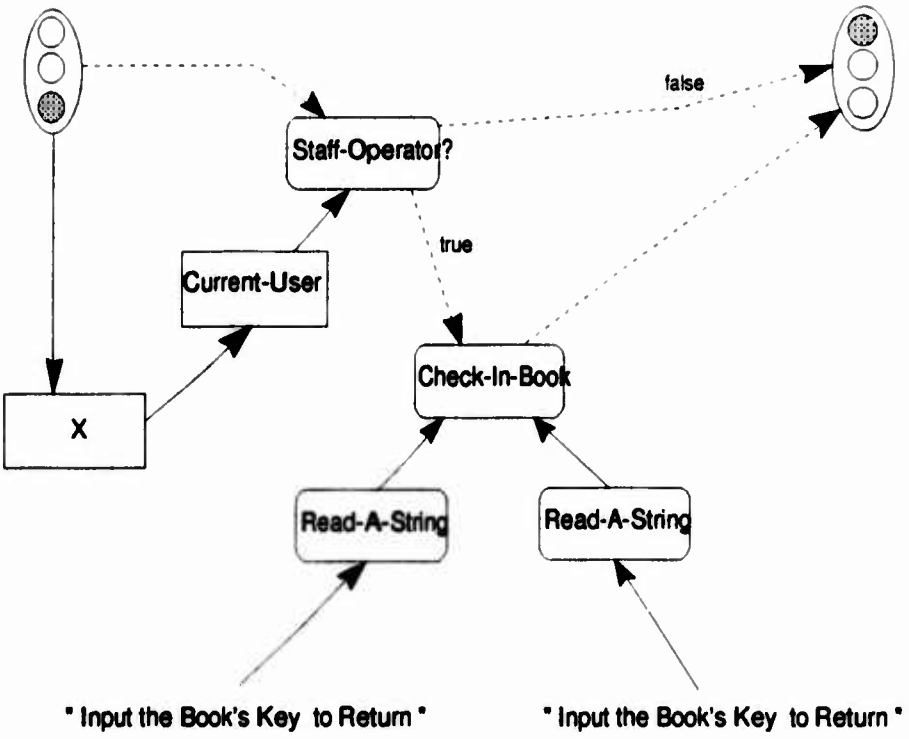


Figure J.30. Visual Refine: Check-Out-Book

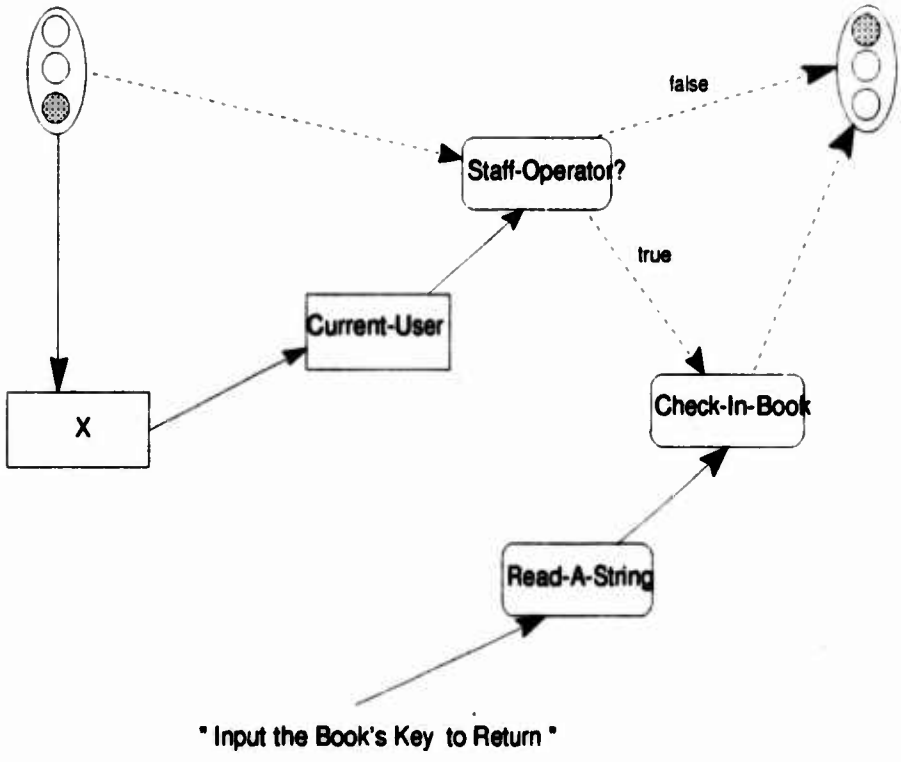


Figure J.31. Visual Refine: Return-Book

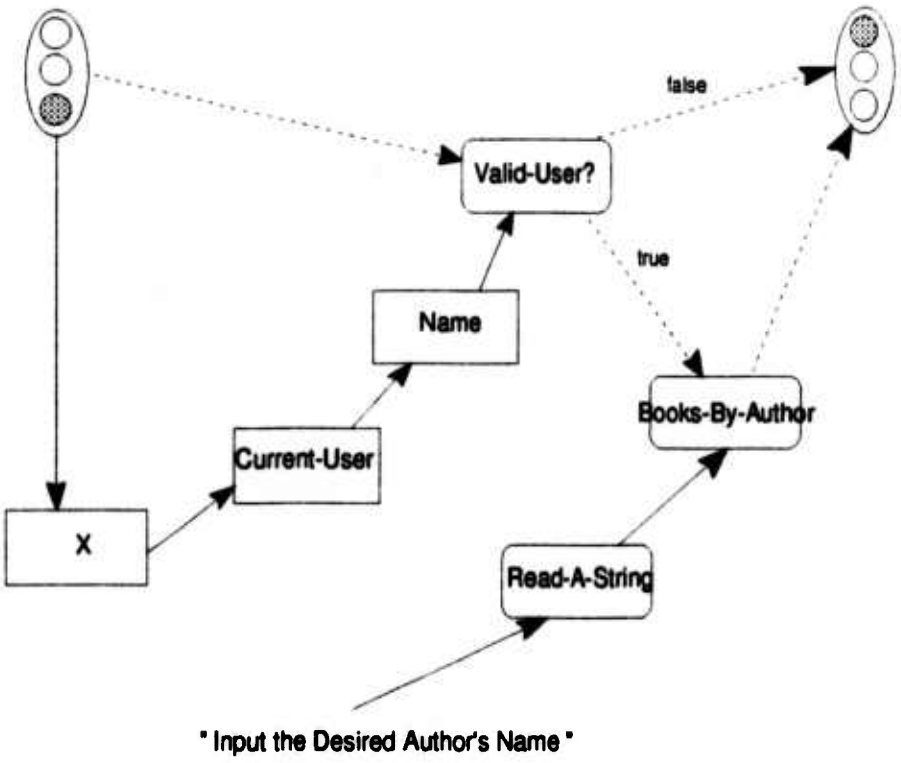


Figure J.32. Visual Refine: List-Books-By-Author

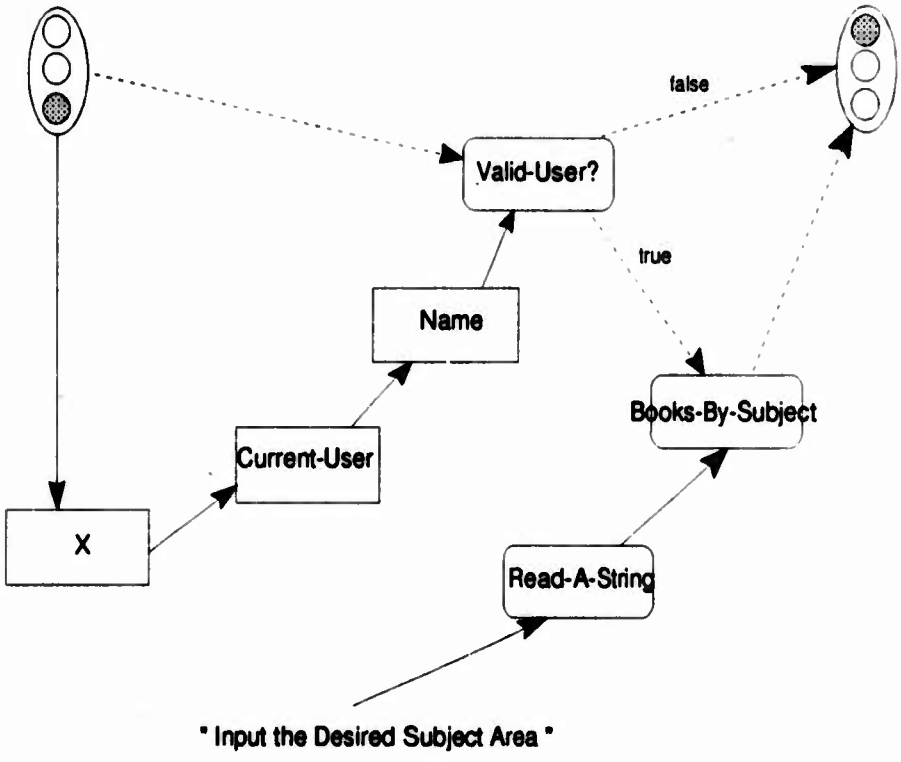


Figure J.33. Visual Refine: List-Books-By-Subject

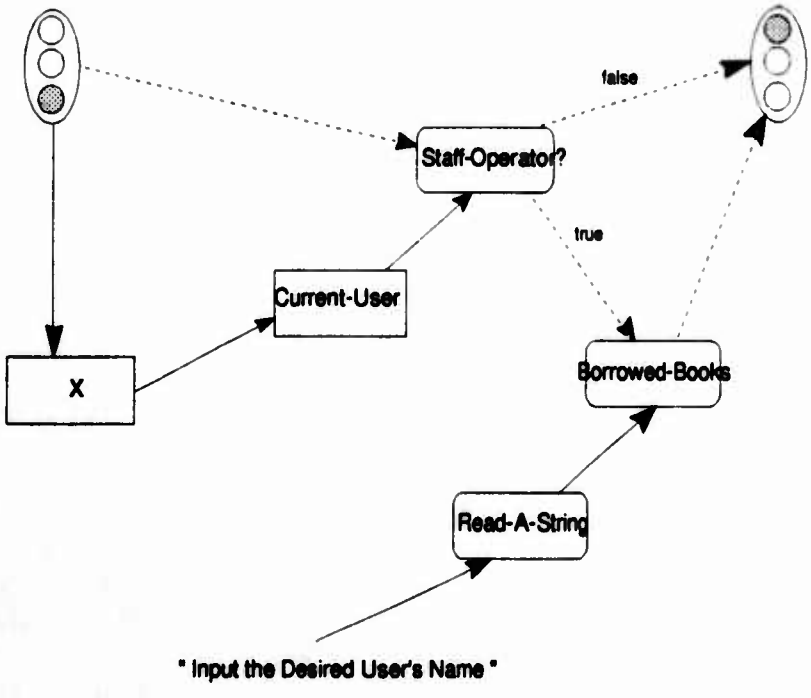


Figure J.34. Visual Refine: List-Borrowed-Books

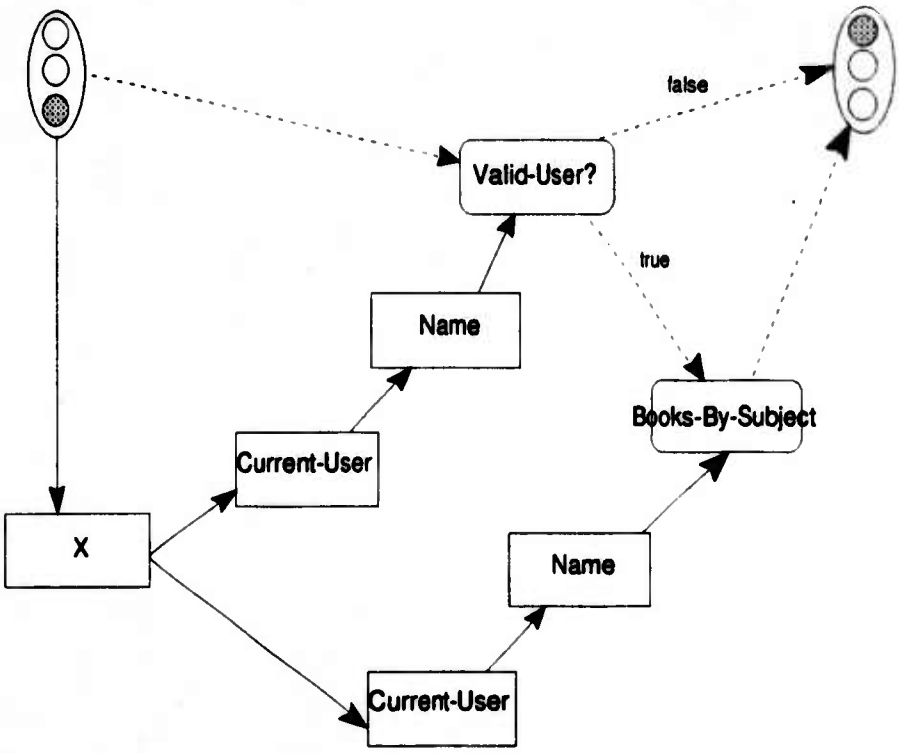


Figure J.35. Visual Refine: List-My-Borrowed-Books

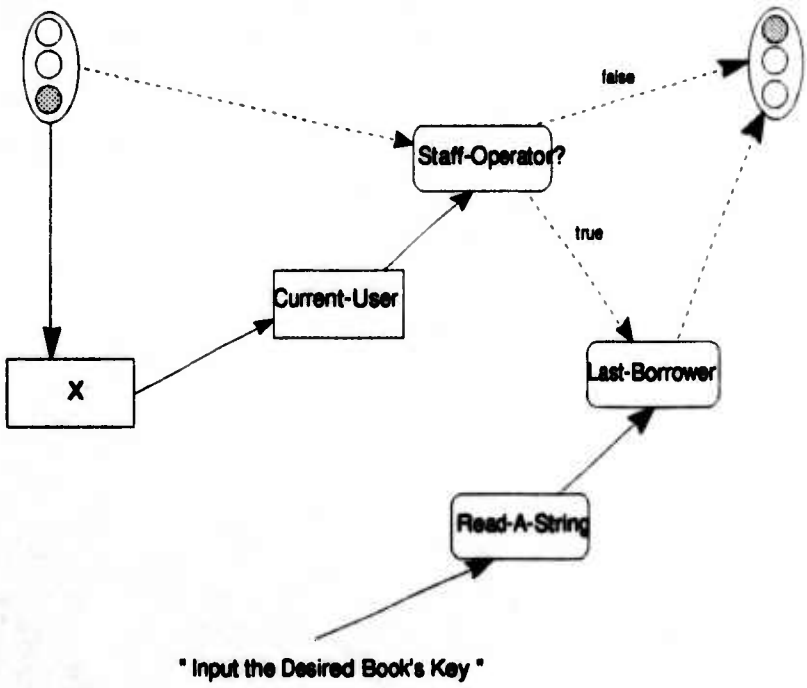


Figure J.36. Visual Refine: List-Last-Borrower

Bibliography

1. *Fourth International Workshop on Software Specification and Design*, 1730 Massachusetts Avenue, N.W. Washington, D.C. 20036-1903: Computer Society Press of the IEEE. April 1987.
2. Bailor, Captain Paul D. *A Theory for Graph-Based Language Specification, Analysis, and Mapping with Application to the Development of Parallel Software*. PhD dissertation, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, September 1989.
3. Bailor, Major Paul D. "Research Area Descriptions Formal Methods Group." Research Seminar class notes, July 1991.
4. Bailor, Major Paul D., et al. "Object-Oriented Database, Software Structural Modeling, and Parallel Simulation Support for the Joint Modeling and Simulation System." Research Proposal, August 1991.
5. Balzer, Robert, et al. "Software Technology in the 1900's: Using a New Paradigm," *IEEE Computer*, *16*(11):39–45 (November 1983).
6. Blankenship, Captain Donald D. *Generalized Method for Transforming Informal Analysis Methods to Refine Formal Specifications*. MS thesis, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1991.
7. Boehm, Barry W. "Improving Software Productivity," *IEEE Computer*, *20*(9):43–57 (September 1987).
8. Booch, Grady. *Object Oriented Design with Applications*. Redwood City, California: The Benjamin/Cummings Publishing Company, Inc., 1991.
9. Davies, Anthony C. "Introduction to Formal Methods of Software Design," *Microprocessors and Microsystems*, *12*(10):547–553 (December 1988).
10. Davis, Alan M. *Software Requirements: Analysis and Specification*. Englewood Cliffs, New Jersey: Prentice-Hall, Inc., 1990.
11. Dromey, Geoff. *Program Derivation: The Development of Programs From Specifications*. Addison-Wesley Publishing Company, Inc., 1989.
12. Greenspan, Sol J. *Requirements Modeling: A Knowledge Representation Approach to Software Requirements Definition*. PhD dissertation, Department of Computer Science, University of Toronto, 1984.
13. Kemmerer, Richard A. "Testing Formal Specifications to Detect Design Errors," *IEEE Transactions on Software Engineering*, pages 32–43 (January 1985).
14. Mills, Harlan D. "Software Engineering - Retrospect and Prospect." In *Proceedings of IEEE (COMPSAC 1988)*, pages 89–96, October 1988.

15. Place, Captain Gene A. *The Development of a Graphical Notation for the Formal Specification of Software*. MS thesis, AFIT/GCS/ENG/90D-07, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1990 (AD-A230458).

16. Protsko, L. Beth, et al. "Towards the Automatic Generation of Software Diagrams," *IEEE Transcation on Software Engineering*, *17*(1):10–21 (January 1991).

17. Reasoning Systems, Inc. *Dialect User's Guide for Dialect^{TM} Version 1.0 on Sun Computers*, July 1990.

18. Reasoning Systems, Inc. *Intervista User's Guide for Intervista^{TM} Version 1.0 on Sun Computers*, July 1990.

19. Reasoning Systems, Inc. *Refine User's Guide for Refine^{TM} Version 3.0 on Sun Computers*, May 1990.

20. Sommerville, Ian. *Software Engineering* (3rd Edition). Addison-Wesley Publishing Company, Inc., 1989.

Vita

Randel Kevin Langloss was born [redacted] He graduated from [redacted] in 1979 and enlisted in the United States Air Force in July, 1979. He served as a C130-H Crew Chief until his acceptance into the Airman Education and Commissioning Program in July, 1983. He entered Colorado State University, Fort Collins, Colorado, in August, 1983 and graduated with the degree of Bachelor of Science in Computer Science in May, 1986. On September 10, 1986 he received a commission in the United States Air Force. After completing required technical training, he was assigned to Headquarters Strategic Communication Division, Directorate of War Planning located at Headquarters Strategic Air Command, Offutt AFB, Nebraska. During this assignment, he was responsible for developing and integrating advanced computer solutions in support of future strategic war planning requirements. He entered the Air Force Institute of Technology in May 1990.

Permanent address: No permanant address available